PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 :
G06F 17/60

A2

(11) International Publication Number: WO 00/54202

(43) International Publication Date: 14 September 2000 (14.09.00)

(21) International Application Number: PCT/US00/06388

(22) International Filing Date: 10 March 2000 (10.03.00)

(30) Priority Data:
60/123,976 11 March 1999 (11.03.99) US

(71) Applicant *(for all designated States except US)*: PAYSYS INTERNATIONAL, INC. [US/US]; One Meca Way, Norcross, GA 30093 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: BLACK, David, B. [US/US]; 41 Clinton Avenue, Ridgewood, NJ 07450 (US). MORGAN, Michael, J. [US/US]; 4220 River Bottom Drive, Norcross, GA 30092 (US). BHAGAT, Rohit [IN/US]; Apartment 1621, 3450 Breckenridge, Duluth, GA 30096 (US).

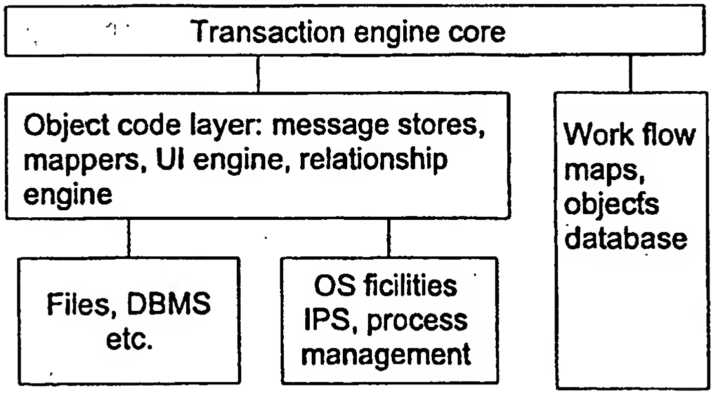
(74) Agents: PRATT, John, S. et al.; Kilpatrick Stockton LLP, Suite 2800, 1100 Peachtree Street, Atlanta, GA 30309–4530 (US).

(81) Designated States: AE, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: METHODS AND SYSTEMS FOR DEVELOPING APPLICATIONS AND FOR INTERFACING WITH USERS

(57) Abstract


Transaction engine core
Object code layer: message stores, mappers, UI engine, relationship engine
Work flow maps, objecfs database
Files, DBMS etc.
OS ficilities IPS, process management

A declarative approach to programming involves providing a transaction engine and a repository. The transaction engine controls the behavior of an application by routing messages between a pre–defined group of work steps. The repository includes a declaration space within which components are defined as well as the relationship between the components. An administrator includes an Integrated Development Environment (IDE) editor for allowing developers to access and modify portions of the repository. In the preferred embodiment, developers are allowed to define attributes of data elements, define relationships between the data elements, and to organize data elements into messages through the repository. Furthermore, developers are able to define the logical–to–physical mapping in the repository, such as by defining message stores. An application is defined by a workflow comprised of a number of worksteps interconnected to each other. The workflow and the worksteps are also defined within the repository. Interfaces developed with the repository and transaction engine can be dynamically changed based on user, security, language, and locale. The repository and transaction engine also enable the inheritance of values and provides message inheritance, language inheritance, sibling inheritance, container inheritance, and attribute inheritance.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania
AM	Armenia
AT	Austria
AU	Australia
AZ	Azerbaijan
BA	Bosnia and Herzegovina
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CU	Cuba
CZ	Czech Republic
DE	Germany
DK	Denmark
EE	Estonia
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GH	Ghana
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IL	Israel
IS	Iceland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgyzstan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakstan
LC	Saint Lucia
LI	Liechtenstein
LK	Sri Lanka
LR	Liberia
LS	Lesotho
LT	Lithuania
LU	Luxembourg
LV	Latvia
MC	Monaco
MD	Republic of Moldova
MG	Madagascar
MK	The former Yugoslav Republic of Macedonia
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
MX	Mexico
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SG	Singapore
SI	Slovenia
SK	Slovakia
SN	Senegal
SZ	Swaziland
TD	Chad
TG	Togo
TJ	Tajikistan
TM	Turkmenistan
TR	Turkey
TT	Trinidad and Tobago
UA	Ukraine
UG	Uganda
US	United States of America
UZ	Uzbekistan
VN	Viet Nam
YU	Yugoslavia
ZW	Zimbabwe

METHODS AND SYSTEMS FOR DEVELOPING APPLICATIONS AND FOR INTERFACING WITH USERS

RELATED APPLICATIONS

This patent application claims priority to, and incorporates herein by reference, co-pending provisional patent application Serial No. 60/123,976 filed on March 11, 1999. Also, this application is related to co-pending provisional patent application Serial No. 60/123,977, filed on March 11, 1999, co-pending provisional patent application Serial No. 60/123,886, co-pending utility patent application filed on March 10, 2000, entitled "Methods and Systems for Performing Workflow" and co-pending utility patent application filed on March 10, 2000, entitled "Methods and Systems for Managing Financial Accounts."

FIELD OF THE INVENTION

The present invention relates generally to methods and systems for developing applications and, more particularly, to systems and methods for providing declarative programming techniques. According to another aspect, the invention relates generally to systems and methods for interfacing with a user and, more particularly, to systems and methods for selectively displaying information to a user.

BACKGROUND OF THE INVENTION

Many companies developed or purchased computer systems years ago, if not decades ago, to manage important data and information. The majority of these legacy systems were developed on mainframes and consequently use mainframe storage technologies, such as IBM Information Management System (IMS), Virtual Sequential Access Method (VSAM), or DB2 relational database management system (RDBMS). Financial institutions, for example, have mainframe applications that track individual customer accounts and over the years have amassed large databases to contain all this information.

Companies cannot readily change or update these legacy systems since they run mission-critical applications. While the migration of these applications to systems having more advanced software and hardware is possible, migration would expose the companies to the risk of a failed conversion. Even though the probability of the risk may be small, a failed conversion can disable one of the mission-critical applications and would be disastrous for a company. It is therefore not surprising that many companies are content to rely on their legacy systems and are cautious in making any update to these systems.

One disadvantage of many legacy computer systems is that by today's standards, they have a fairly limited user interface. To enter data, the user is required to place the data at a certain location on the screen. Similarly, the legacy systems require data output from the system to be sent to a specific location on the screen. For instance, with this type of interface, the name of a cardholder must always be displayed in one location of the screen, the account number in a second location, the remaining balance in a third location, etc. With these legacy systems, ensuring that particular information is displayed in a consistent manner on all screens without special programming efforts is a challenge. This type of interface places a large constraint on how a user interfaces to the legacy systems.

Additionally, this interface is not efficient in the way that a user interacts with the systems. This type of interface does not optimally convey information to the user since it typically presents a large amount of information to the viewer in multiple fields spread throughout the screen. The user also cannot easily enter data since the data must be entered at specific locations on the screen. Further, this type of interface, by itself, is not compatible with displays that allow for dynamic alteration of displays, such as Windows®, and is therefore not user-friendly.

Other limitations exist with the legacy systems and even typical state-of-the-art displays that allow for dynamic alteration of displays, such as Windows®-based applications. One of these limitations is the positioning of elements in the display are defined at fixed positions within a display. Because elements are defined in an absolute manner, developing and maintaining consistent screens becomes very difficult.

Instead of completely migrating a legacy system to a new platform, updates may be made to a legacy system to improve its performance. With regard to the user interface, such updates may involve implementing terminal emulation to provide a more robust interface. In general, terminal emulation involves maintaining the legacy system on the mainframe but providing a client application that emulates the terminal protocols, such as by reading data from and sending data to specific locations on the screen. Screen scraping is a relatively easy and inexpensive way that allows users to interact with a legacy system.

The existing methods of terminal emulation, however, are not without their shortcomings. Terminal emulation, for example, typically separates the front end client application from the mainframe programs and data files and, as a result, slows down communications and degrades the performance of the overall system. Although screen scraping is inexpensive and easy, screen scraping still requires the user to move between multiple screens to perform standard functions. Thus, a need still exists for a better way of interacting with legacy systems.

Many interfaces to computer systems, not just those having legacy systems, are rather inflexible in the way that information is displayed. For example, a common way to display data is to place the name of a parameter, such as customer account number, next to a field containing the actual data, such as the actual account number. While displaying data in fields is convenient, a challenge occurs when certain data cannot be displayed because of security concerns. For example, different users of a system often have different levels of access to the data and some users may not be permitted to see or modify all parameters. The values for these sensitive parameters must therefore be omitted when displaying data to those users who do not have proper authorization. To provide this security, computer systems must have another layer of complexity in the software to ensure that each user has the proper authorization to view the data. This complexity places an added burden on the development, maintenance, and operation of the system.

When data is not displayed because of security reasons, a typical computer system maintains the same template for the display but conceals the underlying data for a certain parameter. For instance, when a user is not authorized to view a customer's Personal Identification Number (PIN), a computer system may display the title "PIN" but gray out or otherwise conceal the customer's actual number for the PIN. This display of the title and field of a concealed parameter is undesirable for several reasons. For one, placing unnecessary information on the display screen wastes the valuable real estate of the screen and is somewhat unsightly. Secondly, the user would likely prefer not to see anything about a field that he or she does not have proper authorization to view. The user, for instance, may not realize that he or she was not granted access to the data and question why that data is not shown. The user may also be resentful that he or she is not entrusted to access the data.

Another way in which computer systems are inflexible is that the interfaces cannot be easily converted to accommodate different languages. For example, a first version of an interface may have the display optimized for one language and many changes may be necessary if a second language is desired. For instance, the translation of text from one language to another may force changes in the spacing allotted for titles of fields, the spacing within each field, the positioning of adjacent fields, as well as other aspects of the interface. These changes are usually implemented by coding an entirely different version of the application or, if a version is multi-lingual, by coding different sets of displays for the different languages. Thus, a significant amount of work is needed to provide interfaces to an application in another language.

In addition to problems with developing and using computer interfaces, many programs are confined to only a certain sized systems. For instance, some programs may only be capable of accommodating a certain number of users or may be limited in the amount of data that may be processed. While these systems may meet the initial needs of a company, the systems may soon become inadequate after a company grows and has demands that exceed the capability of the system. At that time, the company is faced with the dilemma of either upgrading to the next sized system or implementing an entirely new system. In either way, the cost and disruption to a company can be substantial.

These examples of the challenges in developing and making interfaces to computer systems and scaleable systems point to some underlying difficulties inherent in developing all software. Programming has traditionally involved writing code to tell a computer "how" to perform a task. Each task must be "described" to the computer as a combination of instructions from a finite pre-defined instruction set. A programmer must therefore explicitly instruct the computer how to perform a task every time that task is to be accomplished. Many programs therefore contain enormous amounts of code that require a considerable commitment of company resources.

Company resources must not only be devoted when creating new applications but, perhaps more importantly, when changes are being made to those applications. It has been estimated that approximately 10% of the effort in an application is exerted in creating the initial version and the remaining 90% is exerted in customizing the application for a customer or in continually updating the application. One reason that such a large percentage of a company's resources must be devoted to customizing and updating an application is that no two customers are truly alike. One customer may operate on a different platform whereby changes to the code is often necessary to run on that operating system. Other changes may be needed since customers often use different databases, such as SQL, VSAM, or DB2. Some customization may also be needed to add functionality to address a customer's unique needs. Furthermore, updating will undoubtedly occur because of bugs and because of updates in related applications, such as when a new version of an operating system is released.

The enormous size of many programs renders it difficult to make the desired changes to the code. As mentioned above, procedural programs contain the same set of instructions for performing a task each time that the task needs to be executed. A single change to just one task, consequently, can easily translate into numerous revisions to the code. The millennium bug, also known as the Y2K bug, illustrates how a conceptually simple change in the representation of the year can be so difficult to change. Programs that contain the millennium bug were coded, often in COBOL, to represent the year with just the last two numbers of the year, such as "99" for the year 1999. These programs, however, are unable to differentiate between a date in the 1900's with one on or after the year 2000. While changing the representation of the year from a two digit number to four digits is simple to understand, the process of revising a program can be quite complex and burdensome. Programs will almost invariably contain multiple lines of code reciting the year with some programs containing thousands of such lines. To make this simple change, efforts must first be directed toward identifying all lines that contain the year followed by efforts to revise each of those lines. These efforts place a tremendous cost both in time, money, and lost opportunities, on the vendors and/or on the end-users of the programs. As should be apparent from the difficulty encountered with the millennium bug, other changes to programs can deplete an equal amount, if not more, of a company's resources.

Advances in programming techniques and languages have alleviated to some degree the difficulties in developing and maintaining applications. For example, in contrast to earlier procedural programming techniques such as COBOL, object-oriented programs reduce the burden on programmers through the concepts of abstraction, polymorphism, inheritance, and encapsulation. In general, programming with an object-oriented language involves identifying the problem to be solved, identifying objects needed for the solution, identifying messages to be sent to the objects, and creating a sequence of interactions among the objects to solve the problem.

A significant feature of object-oriented programming is that the objects can be reused and can form the foundation for other applications. Class libraries that are available on the market allow a great deal of abstraction and reduce the amount of code that must be written. Studies have demonstrated that programmers not having access to those classes spend a great deal more time in writing a lot more code than programmers who rely on those classes.

The classes that are currently available are generally very low-level classes and cover a broad range of horizontal applications. While the classes do reduce the amount of programming necessary, a substantial amount of work is involved in building a set of application-oriented classes on top of that foundation. Successful projects therefore still end up spending a great deal of time developing elaborate and appropriate classes, which can play many roles in building a particular application.

Despite the use of object-oriented programming and classes, applications still require a considerable amount of code that must be written, debugged, and updated. One reason for the heavy reliance on code is that there is a tremendous need in the marketplace for customization of applications. Many of the earlier object-oriented programs are not very configurable. After the object-oriented analysis has been performed and the classes are built and defined, the programs are still characterized by having behaviors hard-coded into the objects. As a result, whenever any change is needed, such as in a behavior or attribute, the change is implemented in essentially the same way that changes have been made for decades, which is by accessing the source code and making the change. Although object-oriented programming techniques have provided for some abstraction, applications are still unable to rely upon higher levels of abstraction and, as a result, many changes are still hard-coded into the class definitions.

Another reason why object-oriented applications are still heavily dependent upon large lines of code is that the applications have redundancies spread throughout the coding. For example, a payment processing system that is used to manage credit card accounts may have a class defined for Transactions, a class for Accounts, and a class for Cards. The Transaction class would have a definition for Account Number, the class for Accounts would also include a definition for Account Number, and the Card class would also have a definition for Account Number. The program is therefore redundant because it contains these multiple definitions for Account Number.

Object-oriented programming is still heavily dependent upon hard-coding in other ways as well. For example, a table having a collection of data values can be represented in the program in various ways depending upon a method selected for implementing it. A very small table may be represented by enumeration statements in C++ or level 88's in COBOL, a larger table with a table of constants compiled into the program, an even larger table might be out in another file, and yet an even larger table might be in a database table. Thus, depending upon the selected method, the syntax of the program would be markedly different. Each place that a table is referenced, the particular syntax for that table appears. Thus, the application contains multiple references to the same table. The need to place the table syntax throughout the program not only is a burden during the development of the application but also renders it difficult to make any changes to the table. To make a change, such as to how data is stored, the program must be changed every place where that table is referenced. Traditional object-oriented programming techniques therefore still often require an overabundance of code.

In addition to being redundant, object-oriented programs are limited in the extent that programs are designed and implemented on a logical level. For example, when accessing databases, object-oriented programming does permit one to design at a logical level but, when implementing the design, the programs contains code, such as SQL statements, for the physical representation of the database. Consequently, during the logical-to-physical mapping, if a change is desired to either normalize or de-normalize some tables, then all of the select statements that refer to those tables have to be changed to represent the physical representation of the table. As another example, at the logical level, a join may exist between two tables with the join being on a particular field. On the other hand, when writing the code, there is no easy way of expressing that to the database other than by repeatedly restating the fact of the join, which fields are used, and every single statement that refers to the two tables that are joined. As a further example, programs often contain stored procedures in order to use databases in an efficient way. Every stored procedure language, however, is different whereby a greater reliance on stored procedures renders the program less database independent. A need therefore exists for a way of expressing database requirements at a logical level and having the physical mapping be more automatic.

Existing object-oriented programming techniques therefore still have many limitations. These techniques, for instance, require an extensive amount of code to be created during the development of an application and also during any revision to the application. One reason for the large size of many object-oriented programs is that the programs contain redundant lines of code and have behavior hard-coded. Because the cost of developing an application can be proportional to the number of lines of code, development of an application can become extremely expensive. The expense is not limited to just the initial development since programs routinely need to be revised. A revision to the program, even a change that is simple at a logical level, requires rewriting and debugging code and is not easily or inexpensively implemented. A need therefore exists for improved programming techniques.

SUMMARY OF THE INVENTION

The present invention addresses the problems described above by providing systems and methods that implement declarative programming techniques. According to one aspect, the invention includes a transaction engine that controls an overall behavior of a system and a repository that has a declaration space containing definitions and relationships of the various components forming an application. A component that forms a building block for an application is a data element. Each data element is defined by a set of attributes that may be inherited or derived from other data elements and plural data elements may be combined into a group. A message type is an abstract representation of a collection of data elements and may, for instance, define the relationships between data elements. A message store defines the logical-to-physical mapping such as by specifying that a data file is a VSAM file, DB2 file, or other type of file. The repository also includes a definition of a workflow for an application. A workflow comprises a set of worksteps linked to each other with connectors. The transaction engine routes the messages between the worksteps according to the defined workflow.

With the declarative approach to programming according to the invention, applications are developed by defining within the repository the various components and their relationships to each other. Applications can therefore be developed and modified by altering the database without requiring the creation of any code. The ability to modify programs without writing code presents a number of advantages. For one, many changes to a program can be implemented without any recompiling and with nominal testing. Also, changes can be easily performed by the end-user, which may only have a basic understanding of program and who is not a programmer. A platform formed of the repository and the transaction engine greatly reduces the cost of developing and maintaining application. Other benefits and advantages of the declarative programming techniques will become apparent from the Detailed Description of the invention.

The repository and transaction engine allow interfaces to be developed that overcome many disadvantages of existing interfaces. For instance, interfaces can be dynamically altered according to the user, security, language, or locale. With regard to security, when data should not be displayed to a particular user, that portion of the display is omitted and the screen repositions the remaining data elements. With regard to language, the elements of a screen automatically reposition themselves to accommodate a different language. With regard to locale, the interface can be suited for a particular user based on such criteria as language, representation of dates, time, decimal points, and currency. The users may be grouped together and the passwords, security, locale, and language can be set for each user or group of users.

Accordingly, an object of the present invention is to provide systems and methods for programming that implement declarative programming.

Another object of the present invention is to provide systems and methods for programming that reduce the amount of code that must be written in developing applications.

Another object of the present invention is to provide systems and methods for programming that allow changes to an application to be made quickly and easily.

A further object of the present invention is to provide systems and methods for programming that reduce the need for redundant code in an application.

A still further object of the present invention is to provide systems and methods for programming that allow a greater degree of abstraction.

Yet another object of the present invention is to provide systems and methods for programming that eliminates the need to have multiple definitions within an application.

A further object of the present invention is to provide systems and methods for developing platform independent applications.

Another object of the present invention is to provide systems and methods for developing scaleable applications.

A further object of the present invention is to provide systems and methods for developing real time applications.

Another object of the present invention is to provide systems and methods for interfacing with legacy systems.

Yet another object of the present invention is to provide systems and methods for interfacing with computers that allow dynamic alteration of displays.

A still further object of the present invention is to provide systems and methods for interfacing that easily accommodates different languages.

A further object of the present invention is to provide systems and methods for interfacing that can be easily created and modified.

Yet another object of the present invention is to provide development platforms upon which applications can be easily developed.

Another object of the present invention is to provide systems and methods of programming that allow additional types of inheritance.

Yet a further object of the present invention is to provide systems and methods for allowing inheritance of data in objects.

Other objects, features, and advantages of the present invention will become apparent with respect to the remainder of this document.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated in and form a part of the specification, illustrate preferred embodiments of the present invention and, together with the description, disclose the principles of the invention. In the drawings:

Figure 1 is a block diagram of a system according to a preferred embodiment of the invention;

Figure 2 is a diagram of interrelationships between design components forming part of the invention;

Figure 3(A) is an example of an object tab for a component forming part of the invention and Figure 3(B) illustrates a drag and drop feature;

Figure 4 is an example of assigned permission levels to a component;

Figure 5 is an example of a data tab for a user component;

Figures 6(A) to 6(I) are examples of data, message kind, database field, message field, GUI, GUI2, label, bar, and help tabs, respectively for a data element component;

Figure 7 is an example of a data tab for an expression component;

Figure 8 is an example of a data tab for a restriction component;

Figures 9(A), 9(B), and 9(C) are examples of data, reference, and label tabs, respectively, for a message type component;

Figure 10(A) is an example of a data tab for a message store component and Figures 10(B) to 10(G) are examples of a type tab for a message store component illustrating different storage types;

Figures 11(A) to 11(D) are examples of data, stored procedure, result, and label tabs for a query set component;

Figure 12 is an example of query node tree;

Figure 13 is an example of a data tab for a query node component;

Figure 14 is a diagram illustrating a join between two database tables;

Figure 15 is an example of a data tab for a join component;

Figure 16 is a diagram of a sample workflow having three worksteps;

Figures 17(A) and 17(B) are examples of data and label tabs, respectively, for a workstep component;

Figure 18 is a diagram of a sample workflow having worksteps and category components;

Figures 19(A) and 19(B) are examples of a data tab and more tab, respectively, for a category component.

Figure 20 is a diagram of a workflow having worksteps, category components, and connectors;

Figure 21 is an example of a data tab for a connector component;

Figures 22(A) and 22(B) are examples of data and wizard tabs, respectively, for a workflow component;

Figures 23(A) and 23(B) are examples of data and label tabs, respectively, for a language component;

Figure 24 is an example of a data tab for a locale component;

Figure 25(A) is a diagram of an attribute inheritance tree and Figure 25(B) is a diagram depicting an application of container inheritance;

Figure 26 is a view from an editor illustrating language inheritance;

Figure 27 contains tables illustrating message inheritance;

Figure 28 is an example of a sample panel having data elements and groups of data elements;

Figure 29 is a table illustrating the positioning of data elements in a Group 1 container in Figure 28;

Figure 30 is a table illustrating the positioning of the components and groups of components in Figure 28;

Figures 31(A) to 31(C) depict the development of a test panel having various components and groups of components;

Figure 32 is an example of an interface built using a development platform according to the invention;

Figure 33 is a representation of the components and groups of components forming the interface shown in Figure 32;

Figure 34 is a representation of one of the groups of elements listed in the container of Figure 33 along with the individual data elements forming that group;

Figure 35 illustrates the definition of one data element component and its positioning relative to a second component in a vertical direction;

Figure 36 illustrates the definition of one data element component and its positioning relative to a second component in a horizontal direction;

Figure 37 is a representation of a message store component and the various storage types available for selection;

Figures 38(A) and 38(B) are partial views of an interface that the ability of the interfaces to dynamically adjust based on user security; and

Figures 39(A) and 39(B) are partial views of interfaces that illustrate the ability of the interfaces to dynamically adjust based upon user language.

DETAILED DESCRIPTION

Reference will now be made in detail to preferred embodiments of the invention, non-limiting examples of which are illustrated in the accompanying drawings.

I. Overview

According to one aspect, the invention is directed to systems and methods implementing declarative programming. In a preferred embodiment shown in Figure 1, a system includes a transaction engine and a repository. The transaction engine in general governs the basic behaviors of the system and coordinates work flow and the repository is a declaration space for storing object definitions. The system is basically a two-layer architecture in which the transaction engine is the lower layer and objects, data tables, and user functions defined within the repository form the upper layer.

As will be described in more detail below, systems and methods according to the invention overcome many of the disadvantages of existing programming techniques. For instance, applications can be developed without writing any code but instead by defining attributes within the repository. The development time and cost is therefore substantially reduced. Furthermore, changes can be made to an application by changing the definitions within the repository, thereby avoiding any need to create or to debug any code. Because changes to a program are almost inevitable, the invention significantly reduces the cost involved in updating and customizing applications. The invention also permits applications to be truly platform independent and scaleable. Additionally, the systems and methods are completely priority-based in real time to ensure that time-based activities are properly handled. Other advantages and features of the invention will become apparent from the description below.

According to another aspect, systems and methods for interfacing with a computer system overcome some of the disadvantages of existing interfaces. For instance, interfaces according to the invention allow the dynamic resizing and repositioning of displays based on such things as security, language, locality, and user. The interface is platform independent and can be developed and modified easily. Additional advantages and features of systems and methods for interfacing are described in more detail below.

II. Architecture

A. Transaction Engine

In the preferred embodiment, the transaction engine is formed from a set of highly generalized C++ classes. As discussed above, the transaction engine provides the basic behaviors for a system and has a core engine that distributes work among various objects and manages their relationships. The core engine is a set of code that moves messages from sources through processing modules, also called work steps, to stores and also moves messages from stores through processing modules to sinks. With the preferred system, most work is represented by messages that spend time in the message stores, some of which are queues. The work may be performed in interactive way, which requires some interaction with a user, or in a non-interactive way. For non-interactive, the core engine determines which workstep to execute next based on declared priorities of the worksteps. For interactive, the core engine waits for user input before proceeding to the next workstep.

B. Repository

In the preferred embodiment, the repository is a persistent collection of object definitions organized into component dictionaries with these dictionaries collectively forming a library. A dictionary is a collection of components of a similar type, such as all message stores. With conventional programming techniques, the relationships among objects would normally be computed during program execution. In contrast, the repository allows these relationships to be defined in the database and also allows a wide variety of restrictions and security relationships to be declared. The repository is not limited in the types of objects that may be defined and, in the preferred embodiment, includes the following components: application, data element, expression, restriction, message type, message store, join, query, query node, user, language, locale, work step, and work flow.

C. Components

The systems and methods according to the invention provide a set of component type definitions in the form of classes. Objects are named instances of component class definitions maintained within the repository and are defined by a set of attributes and values.

Reuse of object definitions is supported through inheritance of the attribute values. As will be described in more detail, component inheritance according to the invention differs from the type of inheritance found in conventional object-oriented programming languages. The objects support the addition of data members and the overriding of attribute values within inherited objects.

D. Attributes

As described above, the components are defined through their attributes, which is also the case for object-oriented programming. One way in which a component according to the invention differs, however, is that in addition to a set of attributes, the components also have values. One advantage of associating values with the components is that reuse of attribute values can be accomplished through inheritance. Inheritance is defined through a parent-child relationship of the parent's attribute. The inheritance relation is limited to a single parent of a similar component type. An attribute maintains information regarding the inherited state of its value and inherited values reflect changes to corresponding parent component attributes. Non-inherited, also called overridden, attributes are unaffected by parent component attribute changes. The set of attribute data types are defined by built-in C++ types, component references, and component reference collections. The built-in and component reference type attributes resolve inheritance by traversing the inheritance tree during the deployment phase, which is the conversion from a development repository schema to an execution repository schema. The attributes state information is analyzed and, for inherited attributes, the parent component is checked for its value and this process is repeated recursively until a non-inherited value is found. Component reference collection type attributes, on the other hand, maintain only the set of component references defined for the owning component. For example, a parent's collection attribute may have references "A" and "B" and the child's collection attribute has reference "C." The child's collection attribute does not contain references to "A" and "B," although they are inherited from the parent. This inheritance structure is maintained in the repository.

E. Administrator

In general, an Integrated Development Environment (IDE) editor provides tools for interfacing with the repository. The IDE editor provides a user administration tool for allowing a developer to define instances of all objects and permits editing of all objects with minimal time and effort. The user administration tool preferably provides an entry for each object type, such as data element type and message store, and lists all instances of objects within their corresponding object type. Each message type that has been defined would therefore appear under message type. The user administration tool supports typical list operations such as insert, collapse, and drag and drop. When a new instance is inserted, all of the basic attribute names automatically appear with it and are ready to be filled in with values. The user administration tool provides an interface suitable for adding new components or otherwise altering the transaction engine or repository. A localization editor tool allows for the editing of language components and language labels for some components and a workflow editor tool allows for the editing of workflow. Additional explanation and description of the IDE editor is provided below.

III. Components

Each of the various components of the invention forming part of a preferred embodiment of the invention will be described in further detail below. The invention, however, is not limited to these specific components but instead may include variations of these components or other components. Even though each component is addressed separately, a general description of some of the components will nevertheless be provided so as to provide a high level overview.

With reference to Figure 2, the data element forms the basic unit of data in an application. Each data element is defined as an independent entity because it comprises everything needed for displaying, editing, storing, security and integrity whether in storage, in memory or on the screen, and furthermore exhibit inheritance. Data element definitions are gathered together into message types.

Message types are completely independent of methods used to store data, exhibit inheritance, and are recursive. Instances of message types are messages, which are kept in message stores. Message stores may be implemented in a variety of ways, including but not limited to RDBMS tables, text files, indexed files, operating system queues, and buffers.

The repository also defines relationships and restrictions. The join component encompasses a wide variety of information concerning the relationships between data elements and messages. The join component allows fields between tables to be joined and allows data elements to be expressed with reference to other data elements. Restrictions express all the limitations on possible values of data elements. The restrictions inherit from each other and are inherited by the objects to which they are attached. The repository also defines users which are organized into groups and have security relationships that may influence a user's interface. The repository also defines individual worksteps. The worksteps are joined together with connectors to define a workflow. The worksteps and messages may have priorities assigned to them which influence the order of execution for the messages and worksteps.

The administrator includes the Integrated Development Environment (IDE) editor for providing a graphical user interface for creating, editing, and deleting component definitions. With this declarative approach to programming, attribute definitions and other data governing the behavior of an application are contained within the repository. Since most definitions contain references to other components, graphical views provided by the IDE editor can be extremely useful in visualizing the interaction of the components. Thus, the figures contain many views of interfaces provided by the IDE editor both to explain the IDE editor and to explain the components and their relationships to other components.

A. Object

As discussed above, the IDE editor allows attributes of objects to be defined within the repository. Figure 3 is an example of an object tab that allows a developer to specify a unique name to the component and to specify any parent to that object. Additional information that may be specified for any component includes the author who created the component, the version level of the component, and a description of the component. Furthermore, each object can have a security attribute, which will be described in further detail below. The object tab shown in Figure 3(A) is common to all components and each type of component will have one or more additional tabs, which will be described in more detail with reference to each component. The repository editor preferably automatically assigns a component name to an object but this name may be modified by the developer.

A drag and drop feature is preferably used to assign a component object to its parent property. An example of the drag and drop feature is illustrated in Figure 3(B). With the repository editor, a previously defined component object may be selected from an object list panel and dragged to an appropriate property, such as SampleDataElement1 and SampleDataElement2.

B. Security

As mentioned above, security is another attribute that may be defined for each object. The invention preferably provides various levels of security and controls permissions to create, read, update, delete, and execute (CRUDE). Figure 4 provides an example of assigned permission levels to various users. As shown in this Figure, a group of users defined by Admin has permission to create, read, and update the component DE_1 and users defined within group User_1 only have permission to read the component.

C. Users

As discussed above, each component can have security assigned based on the user or group of users. The users, in turn, are defined with reference to a user component. An example of a data tab for a user component is shown in Figure 5. As shown in the Figure, a user component includes an identification string attribute defining the user, a password string attribute defining the password for the user which is used at login, and a user group attribute to define whether the component is a single user or a group of users. Additionally, a user component includes attributes defining a user's preferred locale and language, languages spoken by the user or group of users, and information attributes Info1 to Info4 designed to hold user-supplied data for security purposes, such as a mother's maiden name of a user.

D. Data Element

A description will now be provided with reference to a data element component and a data element group component. A data element component is used to define all data for an application and contains information relating to both the application and to the user interface. The data element group component is a specific type of data element that allows the combination of multiple data element components and data element groups into a single object. For instance, a data element group can correspond to a physically contiguous grouping of data or to a rectangular area of the display screen.

An example of a data tab for a data element component is shown in Figure 6(A). A data element component includes a data type attribute for defining the internal data type for the component and how its value is stored. The data type may be any type, such as but not limited to character, boolean, integer, real, alpha, alpha-numeric, numeric, data, type, currency, group, bit map, or array. The data element data tab also includes attributes allowing the length of the data to be defined, such as the number of bits required by bit map and array. The data element component, as described above, may be a group of data elements with the various elements in the group being defined within the elements attribute. Also included in the data element data tab are attributes for expressions and restrictions, both of which will be described in further detail below. In general, the restrictions attribute can be used to define restriction objects used to validate data integrity of data elements.

As shown in Figure 6(B), the data element component is also defined through a data element message tab. Through an audit attribute on this tab, an audit can be defined so as to be generated any time the field is updated. Figure 6(C) shows an example of a database field tab which allows a developer to define attributes on how to store data elements, such as within an RDBMS or VSAM message store. An example of a data element message field tab is shown in Figure 6(D). Through this tab, a developer can define the external data type of the component and the format of the components data within a message type data stream. The IDE editor preferably provides a drop-down box containing the possible selections, such as but not limited to native, ASCII, EBCDIC, unsigned BCD, signed BCD, or bit field.

An example of a GUI tab is shown in Figure 6(E). In general, the GUI tab attributes allow a developer to define how the data element will be graphically represented on a display panel or screen. The view attribute defines the type of control used to view the data value of a component. A position attribute defines the location of the component within its group if the element forms part of a container or panel. The position attribute preferably allows the component to be displayed at a fixed location on this screen, such as top left, top right, bottom left, or bottom right, or to display the component relative to another object. A panel type attribute defines the type of window used to display the panel.

An example of a data element GUI2 tab is shown in Figure 6(F). The GUI2 tab provides additional attributes for defining aspects of a data element. These attributes include expression attributes for defining how the element is displayed and an actions attribute that defines how an application will respond to predefined panel events.

An example of a data element label tab is shown in Figure 6(G) and includes attributes for defining a label for the component. An example of a data element bar tab is shown in Figure 6(H) and includes attributes for defining such things as a tool tip, status bar, menu bar, and tool bar. An example of a data element help tab is shown in Figure 6(I). The help tab allows a developer to define the text that is displayed when a user selects a help function.

E. Expressions and Restrictions

As discussed above, an expression may form part of the definition of an object. Expressions may be used in many places for various purposes, such as to assign and analyze data, invoke functions, or define symbolic names. An expression is defined by an expression component and is defined preferably using C or C++ read-only syntax. A sample data tab for an expression component is shown in Figure 7(A) and defines the value of the data element and provides a default value for the data element. Both of these attributes are defined with reference to an expression component that would be inserted in those fields. Figure 7(B) illustrates an example of an expression edit panel for defining an expression. The expression component also includes an attribute for defining the language for the text. The language component will be described in further detail below.

A restriction component is a boolean expression that provides constraints on data element values and also integrity tests for message types. Using the drag and drop feature, multiple restriction objects can be assigned to the restrictions (RR) property of both data elements and message types. An example of a data tab for a restriction component is shown in Figure 8.

F. Message Types and Message Stores

A message is a mechanism used to input, output, and transfer information throughout an application, and a message type component defines a physically contiguous group of data items. A message instance is an actual message for a particular message type. In other words, if a message type is defined to hold an account number and name as data elements, then a message instance of that message type might hold the values 1234 and John Doe. A message instance result set is a special class of a message instance that contains multiple message instances. For example, if a message instance contains a single account number and name, a message instance result set would contain multiple account numbers and names.

An example of a message type data tab is shown in Figure 9(A). A priority attribute is provided on the message type data tab and allows a developer to assign an integer value to the message type. The integer value is used by the transaction engine to set an overall priority level for processing. In addition to messages, an application component, workflow component, and workstep component also have an associated priority attribute used by the transaction engine in determining priorities of these elements. Also included on the message type data tab is an items attribute. The items attribute allows a developer to define the data elements and/or data element groups that combine to make up the message type. The message type data tab also includes a timed attribute that allows a developer to define a timing requirement associated with the message type. Additionally, the user can define an appropriate response if the message type is not completed within the allotted time period.

An example of a message type reference tab is shown in Figure 9(B). The message type reference tab has a restriction attribute that allows a developer to define the restrictions used to validate the data integrity of the message type data stream. These restrictions are executed by the transaction engine during parsing of the input data stream. An example of a message type label tab is shown in Figure 9(C). The message type label tab has a group label attribute which is an expression that allows a developer to define the label that will appear in a group of messages of this message type in an explorer control for the language specified. For example, if this message type represents a customer, the attribute may be set to "Customers." The message type label tab also includes a single label attribute which is an expression that allows a developer to define the label that will appear for a single message of this message type in an explorer control for the language specified. For example, if this message type represents customers, a label such as "Customer: Big Spender" may be displayed. While the group label attribute will often be a simple character expression, the single label attribute will likely take advantage of the fact that this is in fact an expression and will combine static text and values that exist in the context.

In general, a message store defines the type of storage used to maintain a message store component. Each message store component has an associated message type that specifies the content and layout of the data written to and read from the message store. Once defined, message stores allow the application to input and output data without regard to the method used to physically maintain the information.

An example of a message store data tab is shown in Figure 10(A). The message store data tab has a Windows® MsAccess® attribute that allows a developer to define the type of access allowed to the message store. The MsAccess® attribute may define various levels, such as read only, write only, read/write, or no access. The message store data tab also has an MType attribute that allows a developer to define the content and layout of all data elements written to and read from this message store and a FileName attribute that allows a developer to define the actual name of the file for a non-RDBMS message store or the table name for an RDBMS message store. The message store data tab also includes an MSShare attribute that allows a developer to define the type of sharing allowed for this message store. This attribute may permit various types of sharing, such as compatible share, no sharing, read share, read/write share, or write share. A timed attribute is also provided on the message store data tab and allows a developer to indicate that timed messages are supported by this message store. Another attribute included within the message store data tab is a temporality attribute that allows a developer to indicate that the message store supports temporal data. In general, temporality allows developers to define the time during which specific values for data elements in the message store are valid. Temporality permits developers to define conditions that become effective for a specified time interval. Also, multiple changes made to the application over a period of time can all be set to become effective at the same time. A TemporalTable attribute allows a developer to indicate that a separate table exists that controls the temporality feature.

Message store components also include attributes for defining the storage type. The invention supports any type of storage including, but not limited to; DBMS, Flat File, In Memory, Queue, Logical, VSAM, HLLAPI and Network. An example of a flat file tab is shown in Figure 10(B) and allows a user to define the type of data within the flat file, such as ASCII or binary. The flat file type tab also includes an OpenMode attribute that allows a user to define the manner in which the file should be opened if it can be written to, includes an append option for specifying that the file should be opened in a manner in which any data written to this file is appended to existing data, and a truncate option which indicates that the file should be opened in a manner in which any existing data in the file is removed before any new data is written to it. An example of a DBMS type tab is shown in Figure 10(C). This tab includes a dbName attribute which allows a developer to define the name of the database on the server, a dbmsType attribute which allows a developer to define the type of server, such as a SQL server, and a dbServer attribute that allows a developer to define the name of the server containing the database. An example of a VSAM type tab is shown in Figure 10(D). By specifying the message store as a VSAM type, applications developed with the invention can access VSAM files on a host computer for data storage.

An example of a network type tab is shown in Figure 10(E). By specifying the storage type as network, a user can define applications that use a network connection for data storage. The network storage type tab includes a transport attribute which allows a developer to identify the type of connectivity that will be used by this message store. The invention supports any type of transport, including but not limited to TCP/IP, NetBios, Lu2.0, Lu6.2, Async, Spex/Ips, Bisync, X.25, Pipe, and File. Other attributes provided on this type tab allows a developer to define the protocol, to select raw or extended connectivity, to define compression, to define a specific port, to define a host address, and to define time out functions.

An example of a HLLAPI storage type tab is shown in Figure 10(F). This type of message store allows a user to define a HLLAPI connection for data storage. This tab includes a settle time attribute for allowing a developer to define the amount of time to wait for a screen to display before electronically reading or scraping the screen, an HLLAPI time out attribute for allowing a developer to define the amount of time to wait for the host to respond before returning an error, and an emulator type attribute to indicate the name of the software package being utilized for HLLAPI. An example of a Disk Storage Location (DSL) storage type tab is shown in Figure 10(G). The DSL type tab allows a developer to define DSL files and to identify a server for the DSL files.

G. Queries and Query Nodes

A query set component provides a way of retrieving or deleting data from one or more message stores by defining the data elements and any limiting criteria. An example of a query set data tab is shown in Figure 11(A). The query set data tab includes a scope attribute which allows a developer to define where the results should be placed in the context, which will affect when the results are removed. The scope attribute in the preferred embodiment includes selections for workstep, transaction, workflow, and application. The query set data tab also includes an action attribute which allows a user to define the initial action to take when executing a command. Through the action attribute, a developer can specify that the initial action should be taken in order to select or retrieve data from a message store or the action should be taken so that data defined by the select criteria is deleted. The query set data tab also includes a TimeoutAction attribute that allows a developer to define the action to be taken if the current transaction reaches a time out.

An example of a query set stored procedure tab is shown in Figure 11(B) and allows a developer to identify a pre-defined procedure that executes and returns desired results. The query set stored procedure tab also includes an input parameter attribute that allows a user to define all input data required by a specific stored procedure in order to execute.

An example of a query set result tab is shown in Figure 11(C). The query set result tab includes a SelectCriteria attribute that allows a developer to define an expression that identifies a set of results. For instance, the select criteria attribute may be used to select all of the accounts for a certain customer, all of the accounts for a currently selected customer in a certain context, or all of the accounts that have a balance greater than a certain dollar amount. A result attribute allows a developer to define data elements to retrieve. A developer is able to select only certain data elements within a message store or to select the entire set within the message store. The query set result tab also includes a sort attribute that allows a developer to define the sort order of the results to be returned.

An example of a query set label tab is shown in Figure 11(D). The query set label tab includes a group label expression attribute that allows a developer to define the label that will appear if a group of messages of this message type are displayed in an explorer control for the language specified. The query set label tab also includes a single label expression attribute which allows a developer to define the label that will appear for a single message of this message type in an explorer control for the language specified.

A query node component provides a way of retrieving data using either a query set or a join component. Each query node can also be assigned a position within a tree-like structure of query nodes and is executed in a sequence as defined by the tree. This provides the ability to link any combination of query sets and joins in an ordered sequence for execution. For example, Figure 12 illustrates a query node structure made up of eight nodes. When called by an application, query node 2 QN2 would first execute its query set or join and retrieve the results and then call both query nodes 5 QN5 and 6 QN6. The data retrieved by query node 2 QN2 would be available to both query nodes 5 QN5 and 6 QN6 and could be used in the definition of their query set or join.

An example of a query node data tab is shown in Figure 13. The query node data tab includes a query attribute that allows a user to retrieve data as specified in a query set object. The query node data tab also includes a node child attribute that allows a user to define another query node as a child of this node in a tree-like structure, such as the one shown in Figure 12.

H. Join

A join component defines a logical link between two tables and a database. The joins, as discussed above, are used by queries to quickly establish a unique row of data that will satisfy specific search criteria. For example, Figure 14 illustrates how two database tables can be joined by the state code and state fields. Using this logical connection provided by the join definition, queries to the database could quickly retrieve the full state name from the first table for any corresponding state code found in the second table.

An example of a join data tab is shown in Figure 15. The join data tab includes a RelType attribute that allows a developer to define the relation type, examples of which are for joining database tables, inheritance, and GUI. The join data tab also includes a parent cardinality attribute and a child cardinality attribute. The parent cardinality attribute allows a developer to define the cardinality rules from the parent to the child table and, conversely, the child cardinality attributes allows a developer to define the cardinality rules from a child to the parent table. These rules, for instance, may include one to none, one to one, or one to many. The join data tab also includes an attribute to allow a developer to identify a message store as the parent and an attribute for allowing a developer to identify a message store as the child. Furthermore, the join data tab includes attributes for allowing a developer to define parent data elements and child data elements.

I. Workflow and Worksteps

As discussed above, the transaction engine routes messages through worksteps to achieve a desired workflow. Each workflow is made up of interconnected units of work called worksteps. Each workstep component defines specific units of work required to perform the overall function of the workflow.

A first step in defining a workflow is to create the individual worksteps. Figure 16 illustrates a diagram of the first step in the formation of a workflow. Figure 16 illustrates three types of worksteps: a source workstep, a mapper workstep, and a sink workstep. In general, a source workstep inputs data from message stores, a mapper workstep provides general data processing, and a sink workstep outputs data to the message stores.

An example of a workstep data tab is shown in Figure 17(A). The workstep data tab includes a step type attribute that allows a developer to define the overall functionality of the workstep component. The workstep may be of any type including, but not limited to, a mapper, a source, a sink, or an interactive workstep that provides a user interface for interactive workflows. The workstep data tab also includes a priority attribute that allows a developer to set the relative priority level for processing this step versus other active worksteps. The workstep data tab also includes a message store attribute for allowing a developer to indicate the message store used as an input for a source-type workstep or as an output for a sink-type workstep, a query node attribute for allowing a developer to indicate the queries that should be executed when selecting data from a message store, and a panel attribute to allow a developer to indicate the panel that should be displayed when processing the workstep.

An example of a workstep label tab is shown in Figure 17(B). The workstep label tab includes a display label attribute that allows a user to have multi-lingual worksteps. With this attribute, a user can identify the name of a specific supported language or a string representation of a label text for a specific language.

After defining the type of the workstep, the developer next defines category components. A category component resides within a workstep and provides a way to accept input messages and produce output messages. Category components are terminal points for all data transfers between worksteps. Figure 18 illustrates the addition of categories to the worksteps of Figure 16. An output category has been added to the source workstep, both input and output categories have been provided to the mapper workstep, and an input category has been provided to the sink workstep. In general a category component represents a scenario that may lead to a specific workstep or provide an exit from one workstep to another workstep.

An example of a category data tab is shown in Figure 19(A). The category data tab includes a category type attribute which allows a developer to define the overall functionality of the category component, such as an input that accepts input messages or an output that produces output messages. A queue attribute provided on the category data tab allows a developer to reference a message store associated with the category component. An example of a category "More" tab is shown in Figure 19(B) and includes an expressions attribute that allows a developer to specify expressions to run sequentially to populate the context of the current category. The category "More" tab also includes a relationships attribute that allows a developer to select data and populate the context of the current category.

After defining the worksteps and the categories, the process of defining a workflow next involves providing connections between the worksteps. A connector component is defined to provide the data path between the output-type category of one workstep with the input-type category of another workstep. Figure 20 illustrates a workflow in which the worksteps and categories shown in Figure 18 have been tied together with connectors. As shown in this Figure, the output-type category of the source workstep is connected to the input-type category of the mapper workstep and the output-type category of the mapper workstep is connected to the input-type category of the sink workstep.

An example of a connector data tab is shown in Figure 21. The connector data tab includes a StepTo attribute that allows a developer to define the workstep that control is to be transferred to when executing this connector and a StepFrom attribute that allows the developer to define the workstep that control is transferred from when executing this connector. The connector data tab also includes a flow attribute that allows a developer to define the workflow to which this connector belongs. The connector data tab also includes attributes for allowing the developer to define the category that this connector should execute in the StepTo workstep and also the category that the connector should execute from in the StepFrom workstep.

A workflow is therefore created by defining worksteps, categories, and connections between the worksteps. An example of a workflow data tab is shown in Figure 22(A). The workflow data tab includes a priority attribute that allows a developer to assign an integer value, such as between 0 and 255, as the priority value for the workflow. This priority value is used to set an overall priority level for processing. The workflow data tab also includes a workflow type attribute that allows a developer to specify when the workflow should execute, such as at start up when the application starts, at shut down just before an application terminates, or at a normal time when the workflow is called. The workflow data tab also includes a steps attribute that allows a developer to identify all of the workstep objects associated with the workflow and a connections attribute that identifies all connector objects associated with the workflow. A flow panel attribute allows a developer to specify a panel, which is a data element group, that is to be displayed when the workflow is running. A champion challenger attribute identifies all champion challenger objects associated with the workflow. An example of a workflow wizard tab is shown in Figure 22(B). The wizard tab allows a developer to add new work steps, new categories, and new connectors to a workflow.

J. Language

A language component, along with locale and user components, gives a developer the ability to generate true international applications. These components enable developers to produce screen labels and help text assigned to specific users. Multiple users can run the same applications simultaneously, with different screen labels and help being displayed based not only on their language, but also on their specific locale.

An example of a language data tab is shown in Figure 23(A). The language data tab includes a month names attribute for allowing a developer to represent long month names and a short month names attribute for allowing a developer to identify the short month names to be displayed. The language data tab also includes a day names attribute for allowing a developer to identify the long day-of-week names to be displayed and a short day names attribute for allowing a developer to identify the short day-of-week names to be displayed. An example of a language label tab is shown in Figure 23(B). The language label tab provides developers with the capability to dynamically switch languages in menus.

A locale component, as described above, along with the language and user components allow developers to produce screen labels based on user language and a specific user locale. An example of a local data tab is shown in Figure 24. The locale data tab includes a language attribute for allowing a developer to specify the language component to use in this locale. The locale data tab also includes attributes for allowing developers to define the character used to separate the decimal point, the character(s) to use to represent numeric and currency groups, the character(s) to use to separate numeric date parts, the character(s) to use to separate numeric time parts, an integer to define the number of digits in each group for non-currency numbers, and an integer to define the number of digits in each group for currencies.

IV. Inheritance

A. Overview

Inheritance has already been described at least in part above with reference to some of the components. In general, multiple types of inheritance exist and the types of inheritance vary from each other according to the way in which they define relationships between components. Inheritance according to the invention differs in many ways from inheritance through existing object-oriented programming techniques.

B. Component/Attribute

One type of inheritance is called component or attribute inheritance. Attribute inheritance defines a relationship between parent and child components where the child receives or inherits the property values of its parent. This relationship is defined through a component's parent property. A parent can have one or more child components but a child component can have only a single parent. An example of an attribute inheritance tree is shown in Figure 25(A). With reference to this Figure, components B, C, and D are childs of component A and each inherits the property values of component A. Further, components E and F are child components of component B and inherit those property values of component B. Similarly, component G is a child of component D. At any point in the hierarchy, inherited property values can be overridden. When this happens, the inherited values of all affected child components are updated to reflect the change made at the parent level. Non-inherited or overridden property values are unaffected by changes made at its parent component. Thus, with reference again to Figure 25(A), all components below component A will inherit its property values. However, if a specific property value is overridden at component B, then components E and F will inherit that property value from component B but all other inherited values will continue to come from component A.

In object-oriented programming, an object accesses another object's data by calling the object's data access methods. Thus, object-oriented programming discourages direct access to common data by other programs. Only the object that "owns" the data can change its content and other objects can view or change the data only by the owner.

In contrast, inheritance with the invention allows components to inherit the data or attribute of its parent. With reference again to Figure 25(A), component E therefore would inherit the values or attributes of component B and component A. The values, however, may be overridden at component B so as to be different from those in component A. For these overridden values, components E and F will inherit the values of component B and not those of component A. Similarly, components E and F may override some of the values inherited from component B.

The ability to inherit attributes or values from other components provides advantages over procedural programming techniques. For instance, an application may define a data element of the year and define its length to be two integers. Thus, the application may represent the year 1999 as "99." Typically, with a procedural program, the developer would have to first locate every line that contains the year and make an appropriate change to the code. With the invention, the developer need only go to the data element defining the year and change the length so that it is four digits rather than just two digits. This change is made within the repository and does not require any recompiling or debugging of the code. Thus, changes can be easily and quickly made to an application thereby reducing the cost to a company.

C. Container

In addition to attribute inheritance, the invention also supports container inheritance.

The transaction engine uses container inheritance to derive some values With container inheritance, a container may be defined to include a group of components. For example, with reference to Figure 25(B), a data element group for A/C History may contain data elements called Balance, Date and Transaction ID. At runtime, it is possible that all contained fields, with the exception of the Balance field, obtain their value from a common message store or message, such as one called Common, in accordance with the application logic. This common source for value may then only be specified with the immediate container of these fields, such as A/C History. The Balance field might refer to an altogether different source for its value at runtime. With reference to Figure 25(B), at runtime, the Date and Transaction ID fields inherit the Data Source attribute, which is that attribute which identifies the location/source of a field's value at runtime, from that of its container, namely A/C History.

D. Language

Many components contain multi-lingual attributes. These attributes contain a value for the attribute for each language supported by the application. Oftentimes, two languages for an attribute can share the same string value. For example, assume an application supports Canadian English, American English, and British English languages. If the application also creates a language called English and makes it the component parent of the other three more specific English languages, then whenever the three specific languages share a string value, the string value can be filled in the multi-lingual attribute for English and the transaction engine will use the English value as the value for any of the other three more specific languages that do not have a value.

Figure 26 provides an illustration of language inheritance in one portion of the display. This portion of the display illustrates American English, Deutsch, and Español. Further divisions may be made within each language, such as European English, British, and Canadian English. Figure 26 also illustrates various data elements and the different languages. For instance, the data element telephone has a label "Telephone" in American English, Canadian English, and European English but has a label "Teléfono" for Español. Similarly, a label named US_Dollars has a label "Currency" for the different versions of English but has a label as "Monedas" for Español. Thus, according to these examples, Canadian English and European English inherit their values from American English but Español has specified values, specifically "Teléfono" and "Monedas."

E. Message

In addition to attribute, container, and language inheritance, the invention also supports message inheritance. Occasionally, a message store may contain messages that are not in the same format, but the messages are part of a family of messages. Individual messages may provide some kind of type field that allows the application to decide how to parse each data element. An example will now be described with reference to Figure 27. Four messages may be utilized to talk to an ATM machine. All the messages contained an account number and the second field describes what the message should do as well as the format for the rest of the message. For instance, when the transaction engine is presented with a "Withdrawal" message, the transaction engine will parse the message by first determining that the message store message type attribute is Root. The transaction will start by parsing in the data element of Root, which in this example will result in CaseAccount and Type. The transaction engine will evaluate the ParseSelector expressions of all children of Root and locate the true expression. In this example, the children of Root are Inquiry and Common and the selector will be true for Common. The transaction engine will next parse the data elements of Common and add "Amount" to the message. The transaction engine will evaluate the ParseSelector expressions of the children of Common and locate the selector of message with "Withdraw" as true. The transaction engine would then be finished parsing since "Withdraw" does not contain any child message types.

F. Sibling

The invention also supports sibling inheritance. As described above, a child component can inherit all attributes or values of its parent component. The child component, however, need not inherit all values and some may be overwritten. Another way in which a child component can inherit from another component is through sibling inheritance. With sibling inheritance, a value in one component is expressed relative to a value in another component. For instance, a first component may have a value that is expressed as two times the value defined in a second component. Other siblings may be added to an existing sibling relationship, such as by specifying that a third component has a value defined relative to the value in the first component which, in turn, is tied to the value in the second component.

G. Value-Based Inheritance

System users can also benefit from inheritance. An Account Master file, for example, could have records organized as an arbitrarily large hierarchy, perhaps corresponding to the organizational structures of various organizations. An agent could set a field value, such as a spending limit, in a record corresponding to a particular organizational level and have it apply to all the people in that organization. The ability to override inherited values would let the agent override that spending limit for any particular group within that organization.

H. Component Replacement Inheritance

Component Replacement Inheritance (CRI) is similar in some ways to normal attribute inheritance. With CRI, however, if a component A' is made to inherit from A, A' will be treated the same as A. As a result, other components or other aspects of an application that refer to A would then use the modified A'. CRI can also chain components together. For instance, a string of components could be defined as $A \leftarrow A' \leftarrow A'' \leftarrow A'''$. With this type of inheritance, A could represent an application delivered by the developer, A' the customization performed by a multi-national company, A'' the customization performed by some national part of the company, and A''' the customization performed by a division within that particular country.

CRI provides many advantages, especially with respect to customization and maintenance. From the developer's viewpoint, the developer can deliver an application and can allow the customer to customize it by using CRI. Rather than needing to modify the application itself, the customer can advantageously use CRI to inherit from the pieces they want to modify. The system treats the modifications as if they occurred within the application itself, but the modifications are isolated in separate files.

V. User Interface

The systems and methods according to the invention allow interfaces to be easily developed. For instance, a data group component D may be defined to include data element components A, B, and C. A data group is not limited to a collection of individual components but may also be defined to include other data groups. For instance, a data group component F may be defined to include the data group component D as well as a data element component E.

The use of these data elements and data element groups allows a developer to easily define a panel on a screen. An explanation will now be provided with reference to the development of a panel shown in Figure 28. This panel includes data elements 1 DE1 and 2 DE 2 that are defined within a panel labeled Group 1. The Group 1 panel also includes a component labeled Button 6. The Group 1 panel forms a part of a larger group labeled Panel 1. The Panel 1 container, in addition to the Group 1 panel, is defined to include additional components labeled Button 1 to Button 5.

An example of a table listing the positions of each element in Group 1 is shown in Figure 29. As shown in this Figure, data element DE 1 is positioned at the top left of the panel, data element DE 2 is positioned relative to data element DE 1 in a vertical direction, and Button 6 is positioned at the bottom right of the panel. An advantage of such a positioning scheme is that it allows the data element DE 2 to be always positioned below the first data element DE 1. Thus, the second data element DE 2 will dynamically reposition itself within the screen if a change in data element 1 DE 1 affects the amount of vertical space consumed by that data element.

Figure 30 is a table that illustrates the relative positioning of the components forming the Panel 1 group. As reflected in this table, the Group 1 panel is at the top left of the display, the first button Button 1 is at the top right, and the third button Button 3 is at the bottom left. Thus, the Group 1 panel and the first and third buttons are at a fixed position on the screen. The second, fourth, and fifth buttons, on the other hand, are positioned relative to other components. The second button Button 2 is positioned relative to the first button Button 1 in a vertical direction, the fourth button Button 4 is positioned relative to the third button Button 3 in a horizontal direction, and the fifth button Button 5 is positioned relative to the fourth button Button 4 in a horizontal direction. Thus, Button 2 will always be below Button 1, Button 4 will always be to the right of Button 3, and Button 5 will always be to the right of Button 4.

Another illustration of building panels will be provided with reference to Figures

31(A) to 31(C). Figure 31(A) illustrates a panel labeled "My Test Panel" which may be formed from one data element component. In creating this panel, the data element component may be defined to have a length of 40, a view of entry, a label equal to "Name," and label positioning equal to top. The data element for the label name can then be added to a data element group called "My Panel." The data element group called "My Panel" can have a type defined as group, a view defined as panel, and a label defined as "My Test Panel." This data element group would then contain the data element for "Name" as one of its children.

The developer may then add "City" and "State" to the panel, as shown in Figure 31(B). To form the panel shown in Figure 31(B), a developer can add a second data element component for the city and define the view as entry, the length as 25, the label as "City:", the label position as top and the position as relative to the name data element. The state data element component can also be added to the panel group and can be defined to have a view as ComboBox, label as "State:", label position as top, and position as relative to the city data element component.

In a similar way, additional components or groups of components may be added to the panel. For instance, as shown in Figure 31(C), a set of buttons labeled as "Print," "OK," and "Cancel" have been added to the group labeled "My Test Panel." The Print, OK, and Cancel buttons can therefore be defined as separate data elements and as a command button and can be included within a data element group that is included within the group for the test panel.

An explanation will now be provided with reference to a payment processing system to illustrate some advantages of the invention. Figure 32 illustrates an interface that may be presented to a user, such as a client representative. The interface shown presents information on a particular card-holder's account, such as the card-holder's name, address, list of transactions, balance and interest information, and a set of control buttons.

Figure 33 illustrates a data element group that defines the interface shown in Figure 32. The data element group entitled DemoGroup contains elements of a NameBlock group component, AddressBlock group component, Transaction group component, Balance component, Interest component, and RSMFormControls group component. The NameBlock group defines the top two rows of the display, the AddressBlock group defines the address information, the Transaction group defines the list of transactions, the Balance element defines the balance, the Interest element defines the interest, and the RSMFormControl group defines the buttons on the bottom of the display. As shown in more detail in Figure 34, the NameBlock data element group includes individual data elements for account number AcctNum, statement date StmtDate, name Name, personal identification number PIN, credit limit CreditLimit, late payment LatePayment, and VIP status of the card-holder VIP.

All of the data elements and data element groups have their positions defined on the interface shown in Figure 32. For instance, looking at the CreditLimit data element shown in Figure 35, the credit limit information is displayed relative to the statement date in a vertical direction. With reference to Figure 32, the credit limit is shown below the payment due date. The position of the StmtDate data element as shown in Figure 36 is defined relative to the PIN data element in a horizontal direction. Thus, as shown in Figure 32, the payment due date field is positioned horizontally to the right of the PIN field. The message store component conveniently allows developers and users to designate the data type. Figure 37 provides an example of a message store component for account balance and illustrates examples of storage types available. These storage types include, but not be limited to DBMS, FlatFile, Inmemory, Queue, Logical, Network, VSAM, HLLAPI, and DSL.

As described above, interfaces developed with this invention allow for security to be assigned based on the user. As an example, Figure 38(A) illustrates account information that may be available to a first user having a greater degree of permission than a second user, which receives the account information shown in Figure 38(B). Advantageously, fields that are omitted when displaying the information to the second user, namely the Open-to-buy and Credit limit information is not shown in the interface displayed in Figure 38(B). Furthermore, the interface is dynamically altered so that the Total accounts information is displayed immediately below the Amount due data. In other words, the interface shown in Figure 38(B) does not include a space between the Total accounts and the Amount due for the omitted Open-to-buy and Credit limit fields. The interface as shown in Figure 38(B) therefore provides no suggestion to the second user that information has been concealed.

Another benefit of the invention is that the interfaces can be dynamically changed according to the language displayed. An illustration of this ability will now be described with references to Figures 39(A) and 39(B). Figure 39(A) illustrates a portion of an interface displayed in an English language and depicts a Transaction field above a Short Name field, an Organization field to the right of the Transaction field, a Logo field to the right of the Organization field, and a Status field below the Logo field. As is apparent from Figure 39(B), the relationships between these fields remain the same even when the interface is displayed in another language. Thus, when the interface is changed to Spanish, Transacción is still above the Nombre Corto field, the Organización is still to the right of Transacción, the Logo is still to the right of Organización, and Estatus is still below the Logo field. The resizing of the interface is also apparent from the tabs.

VI. Scalability and Other Benefits

The systems and methods according to the invention define a platform that presents an environment specially tailored to efficiently support financial programming. Typical financial programs repeatedly use the same types of constructs, such as varying descriptions of account numbers or embedded SQL calls to a database. By applying object-oriented programming principles to abstract these common constructs into platform objects, the invention provides a mechanism to reduce the size and scope of the non-common, application-specific software and thereby reduce maintenance costs.

In addition to object-oriented techniques, the invention employs a non-procedural programming style. With the invention, developers declare what the application should do, as opposed to the more traditional development model where the developer creates a program that tells how to realize an application. To support this non-procedural or declarative method of programming, the invention provides a relatively small number of components and component properties, also called attributes, that an application developer can define. The platform according to the invention understands how a component should behave based on what the application developer has defined the attributes. The declarative approach provides a way to push common programming into the platform, reduce the size of the application-specific portion, and reduce the associated maintenance costs.

The platform according to the invention is declarative and object-oriented in nature. The platform is built not only using an object-oriented language, such as C++, but makes objects the basis for the platform environment itself. A significant advantage achieved from this approach is to make systems built using this platform easier to extend and maintain.

The invention also presents advantages in inheritance. In general, inheritance provides the ability to define a hierarchy such that objects lower in the hierarchy can automatically "inherit" values associated with objects above them. A main advantage of inheritance is the ability to make a change in a single place that should apply to a collection of objects. For instance, one can define a base AccountNumber data element and have all Account Number definitions inherit from that base. Assume that all account numbers start out being a maximum of 15 digits long, and one encodes this information in the base AccountNumber definition. If there is a subsequent need to expand the maximum length of account numbers to 21, one need only make the change in one place and have it apply to all my account numbers. Similarly, if a set of related Message Stores inherits from a base Message Store definition that initially associates the store with the Microsoft SQL Server RDBMS, changing the base definition to use Oracle will make all inheriting definitions use Oracle.

With procedural programming, the computer knows how to do a small number of very basic commands, such as assignment, branching, and arithmetic. The developer uses the commands to tell the computer how to perform a task. General purpose programming languages support procedural programming.

Application-oriented programming languages embody application-specific knowledge into a set of higher-level programming constructs. Application in this sense may be application areas, such as financial transactions or word processing. Application-oriented languages may be procedural or declarative; they are declarative if the way they provide access to their higher-level constructs is via interpreting attributes.

Declarative programming can reduce the number of places in a program where a developer needs to make changes. For example, authorization systems keep counts on account activity, such as a daily tally of approved transactions for each account. In a procedural authorization system, there would be n places in that program where it could decide to approve a transaction. To keep the tally, the developer would tell the system in each of those n places to look up the appropriate data element and increment it.

On the other hand, a declarative approach would take advantage of the fact that no matter what the value of the tally is at any given time, it is always the current count of approved transactions. So the developer could declare that definition to be an attribute of the tally data element itself. According to the invention, the tally would be a derived data type. The developer can take advantage of the fact that the platform according to the invention knows how to take counts of records that exist in message stores (optionally partitioned by account and approved/declined).

The declarative approach breaks the direct tie in the code between approving a transaction and updating data elements that care whether it was approved. From a maintenance point of view, if the definition of the tally data element needs to be changed, a change need only be made at the data element itself whereas with procedural applications the n places in the code that contain the definition would first have to be located and then those lines of code would have to be changed.

With the platform according to the invention, applications are collections of workflows and each workflow is a set of interconnected worksteps. Information flows among worksteps as messages. Each workstep is an independent processing unit in that it performs a function driven entirely by its input message(s). A single process could schedule and run the entire application, but the message passing and independent nature of the worksteps provides scalability.

An application can run subsets of workflows as different threads or processes or on different processors. In fact, each workstep could run on a separate processor. In addition, since each processor would essentially be a server taking work from its message-based in box, the same workstep could be assigned to multiple processors so that work destined for a particular workstep could be served by any of the n processors running that workstep.

Ultimately, a set of n available processors may be provided, any one of which could run any workstep that had available work as soon as it became free. Limits, however, could be placed on which worksteps a particular processor would be eligible to run. When a user interacts with the system, for instance, that user's workstation could be the only server eligible to execute the corresponding interactive worksteps.

The forgoing description of the preferred embodiments of the invention has been presented only for the purpose of illustration and description and is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Many modifications and variations are possible in light of the above teaching.

The embodiments were chosen and described in order to explain the principles of the invention and their practical application so as to enable others skilled in the art to utilize the invention and various embodiments and with various modifications as are suited to the particular use contemplated.

CLAIMS

What we claim:

1. A system for use in managing accounts, comprising:

a repository for storing definitions of components, for storing definitions of message types, for storing definitions of worksteps, and for storing a definition of a workflow;

the components include data elements wherein attributes of each data element is stored in the repository;

each of the message types comprising a set of data elements;

each workstep defining a unit of work for being performed by the system wherein multiple worksteps define the workflow for processing account information; and

a transaction engine for passing messages between worksteps according to the workflow.

2. The system as set forth in claim 1, wherein the data elements include attributes for defining a display of information associated with the data elements.

3. The system as set forth in claim 2, wherein the attributes define a position of the information within the display.

4. The system as set forth in claim 3, wherein the attributes define a position of the information relative to the display.

5. The system as set forth in claim 3, wherein the attributes define a position of the information relative to information associated with at least one other data element.

6. The system as set forth in claim 1, wherein the attributes define security associated with the data element.

7. The system as set forth in claim 1, wherein the components include language components for defining a plurality of languages that may be used in an interface to a user.

8. The system as set forth in claim 1, wherein the components include a locale component for defining a plurality of locales that may be used in an interface to a user.

9. The system as set forth in claim 1, wherein the repository includes definitions for a join component, the join component defining a relationship between data elements.

10. The system as set forth in claim 1, wherein the repository includes definitions for a restriction component, the restriction component defining limitations on values of at least one data element.

11. The system as set forth in claim 1, wherein the repository includes definitions for an expression component.

12. The system as set forth in claim 1, wherein the repository includes definitions for a user component, the user component identifying attributes of at least one user.

13. The system as set forth in claim 1, wherein the repository includes a definition for a message store, the message store specifying a storage type for data associated with at least one of the data elements.

14. The system as set forth in claim 13, wherein the message store includes definitions for file format.

15. The system as set forth in claim 13, wherein the storage type includes network storage and the message store includes definitions for transport type.

16. The system as set forth in claim 1, wherein the repository includes definitions for a query component, the query component defining an interaction with data associated with at least one data element.

17. The system as set forth in claim 1, wherein the repository includes definitions

for connectors, the connectors linking the worksteps to form the workflow.

18. The system as set forth in claim 1, wherein the transaction engine is for prioritizing the messages.

19. The system as set forth in claim 1, wherein the transaction engine is for scheduling the messages.

20. The system as set forth in claim 1, further including an interface for use in altering the definitions stored in the repository.

21. A user interface for use with a computer system, comprising:

a repository for storing definitions of components, wherein each component has at least one attribute related to a representation of information associated with the component to a user; and

a transaction engine for representing the information to the user in accordance with the at least one attribute for the component.

22. The user interface as set forth in claim 21, wherein the at least one attribute is a position of the information on a display.

23. The user interface as set forth in claim 22, wherein the at least one attribute defines a position of the information relative to the display.

24. The user interface as set forth in claim 22, wherein the at least one attribute defines the position of the information relative to information associated with another component.

25. The user interface as set forth in claim 21, wherein the attribute defines security restrictions of the user to the information.

26. The user interface as set forth in claim 21, wherein the attribute defines one of a plurality of languages and the transaction engine presents the information to the user in a designated language.

27. The user interface as set forth in claim 21, wherein the attribute defines one of a plurality of locales and the transaction engine presents the information to the user in a format particular to a designated locale.

28. The user interface as set forth in claim 21, further comprising an interface for allowing a user to alter the definitions stored in the repository.

29. The user interface as set forth in claim 21, wherein the information is account information.

30. A user interface for use with a processing system and for displaying account information, the user interface comprising:

a display for displaying account information;

a repository for storing data defining a manner in which the account information is to be displayed;

an engine for receiving the account information and for displaying the account information at the display in accordance with the data stored in the repository; and

an interface for enabling a user to alter the data stored in the repository;

wherein changes to the data stored in the repository result in a change in the manner in which account information is displayed.

31. The user interface as set forth in claim 30, wherein the data stored in the repository includes a language in which the account information is displayed.

32. The user interface as set forth in claim 30, wherein the data stored in the repository includes a locale in which the account information is displayed.

33. The user interface as set forth in claim 30, wherein the data stored in the repository controls security rights to the account information.

34. The user interface as set forth in claim 30, wherein the data stored in the repository defines groups of users.

35. The user interface as set forth in claim 30, wherein the engine is for receiving legacy data.

36. The user interface as set forth in claim 30, wherein the engine is for displaying legacy data in a tabular format.

37. The user interface as set forth in claim 30, wherein the data stored in the repository includes join information for use in defining links to allow a user to navigate through the user interface.

38. The user interface as set forth in claim 30, wherein the data stored in the repository includes expression information for use in selectively displaying items on the display.

39. The user interface as set forth in claim 38, wherein the expression information is for use in selectively displaying menu items.

40. The user interface as set forth in claim 38, wherein the expression information is for use in selectively displaying buttons.

41. The user interface as set forth in claim 38, wherein the expression information is for use in selectively displaying fields.

42. The user interface as set forth in claim 30, wherein the data stored in the repository includes query information for use in building explorer views.

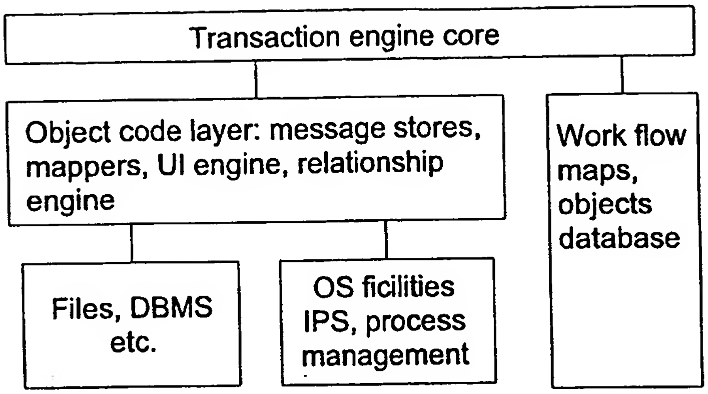

Transaction engine core
Object code layer: message stores, mappers, UI engine, relationship engine
Work flow maps, objects database
Files, DBMS etc.
OS ficilities IPS, process management

FIG. 1

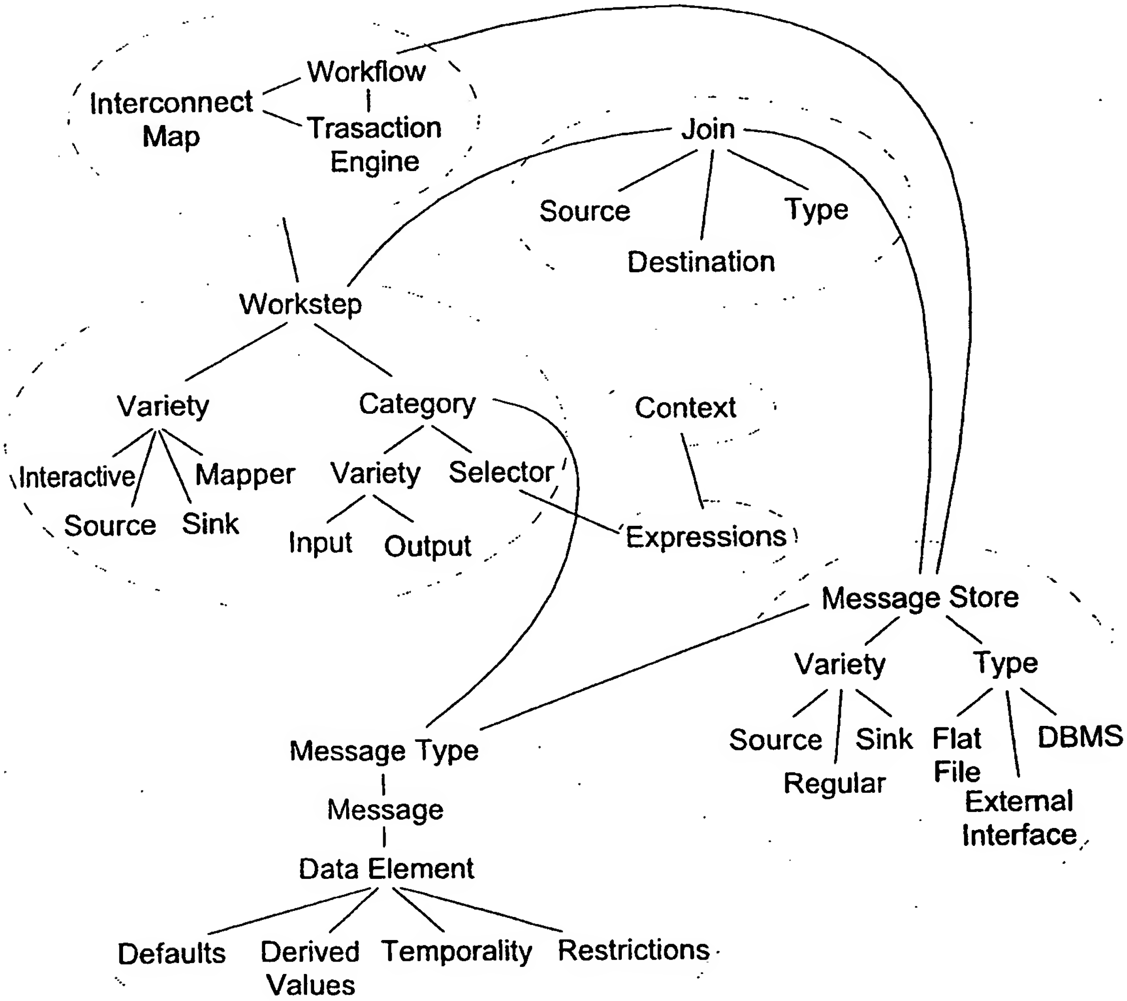
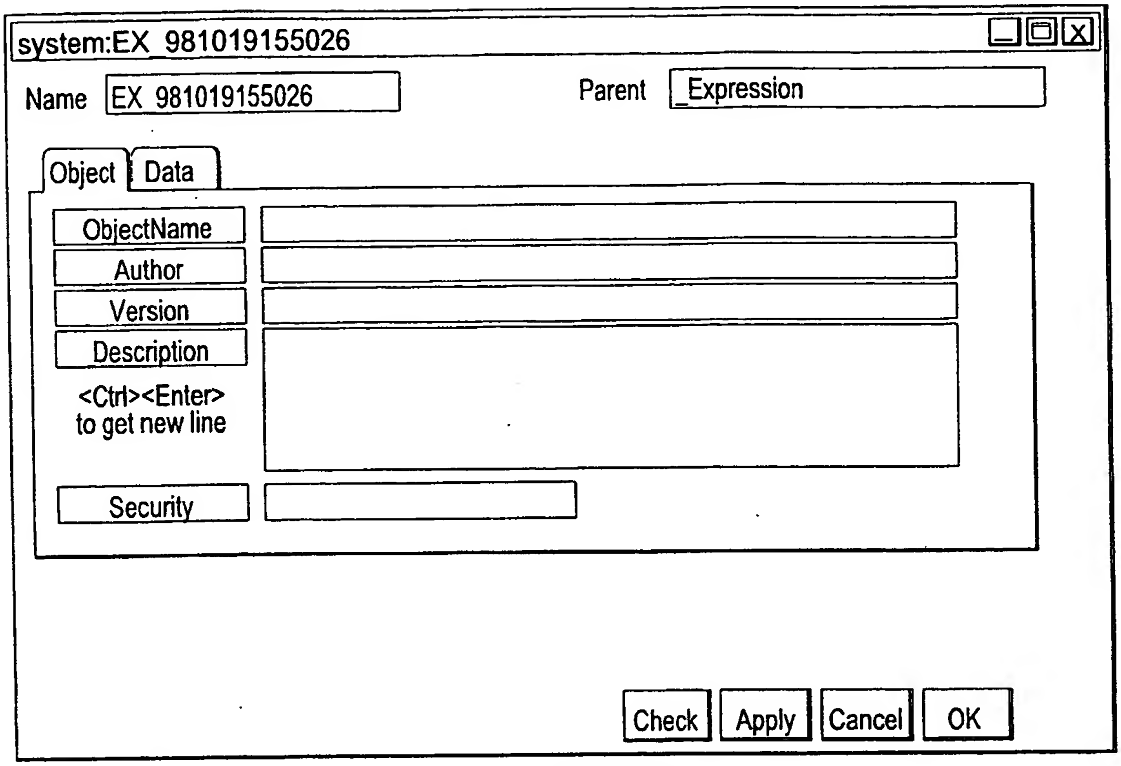

Interconnect Map
Workflow
Trasaction Engine
Join
Source
Destination
Type
Workstep
Variety
Interactive
Mapper
Source
Sink
Category
Variety
Selector
Input
Output
Context
Expressions
Message Store
Variety
Source
Regular
Sink
Type
Flat File
External Interface
DBMS
Message Type
Message
Data Element
Defaults
Derived Values
Temporality
Restrictions

FIG. 2



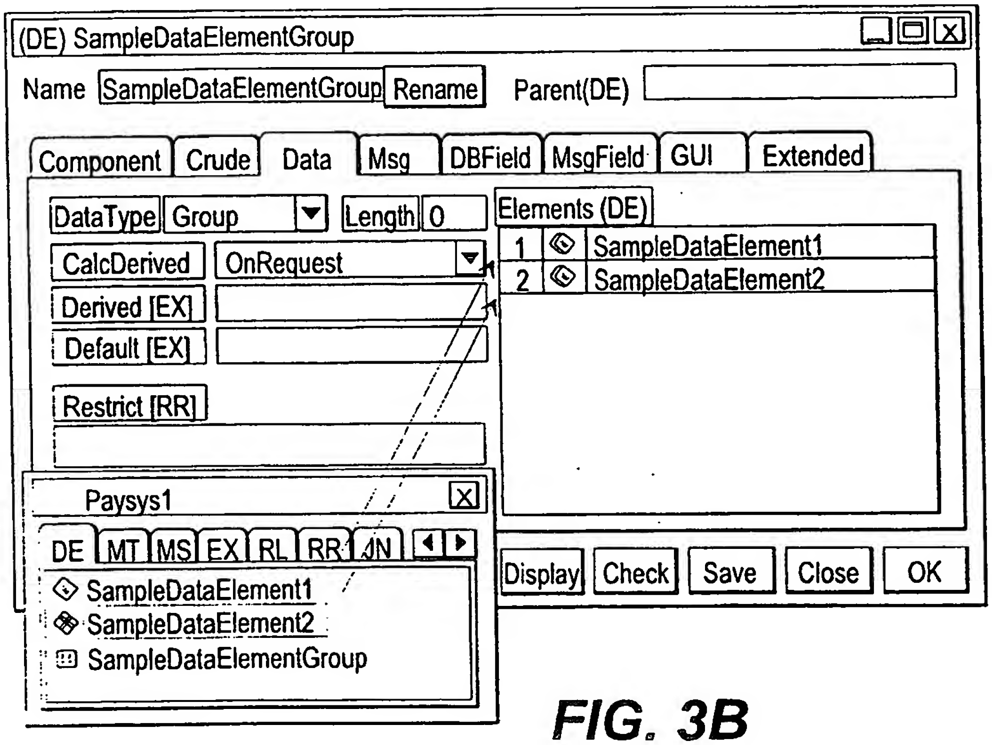
system:EX_981019155026

Name EX_981019155026

Parent _Expression

Object Tab

FIG. 3A



(DE) SampleDataElementGroup

Name SampleDataElementGroup

Parent(DE)

DataType Group

Length 0

CalcDerived OnRequest

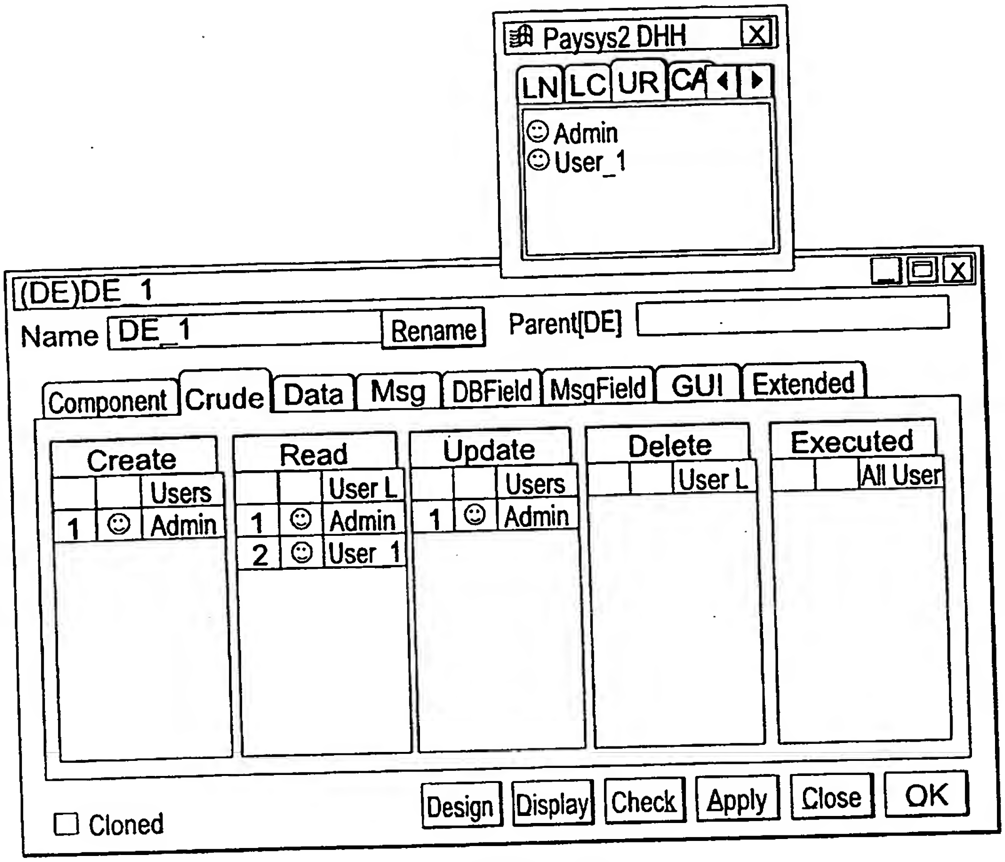
Elements (DE)

1	SampleDataElement1
2	SampleDataElement2

Paysys1

- SampleDataElement1
- SampleDataElement2
- SampleDataElementGroup

FIG. 3B



☺ Admin
☺ User_1

Name DE_1

Parent[DE]

Create		
		Users
1	☺	Admin

Read		
		User L
1	☺	Admin
2	☺	User_1

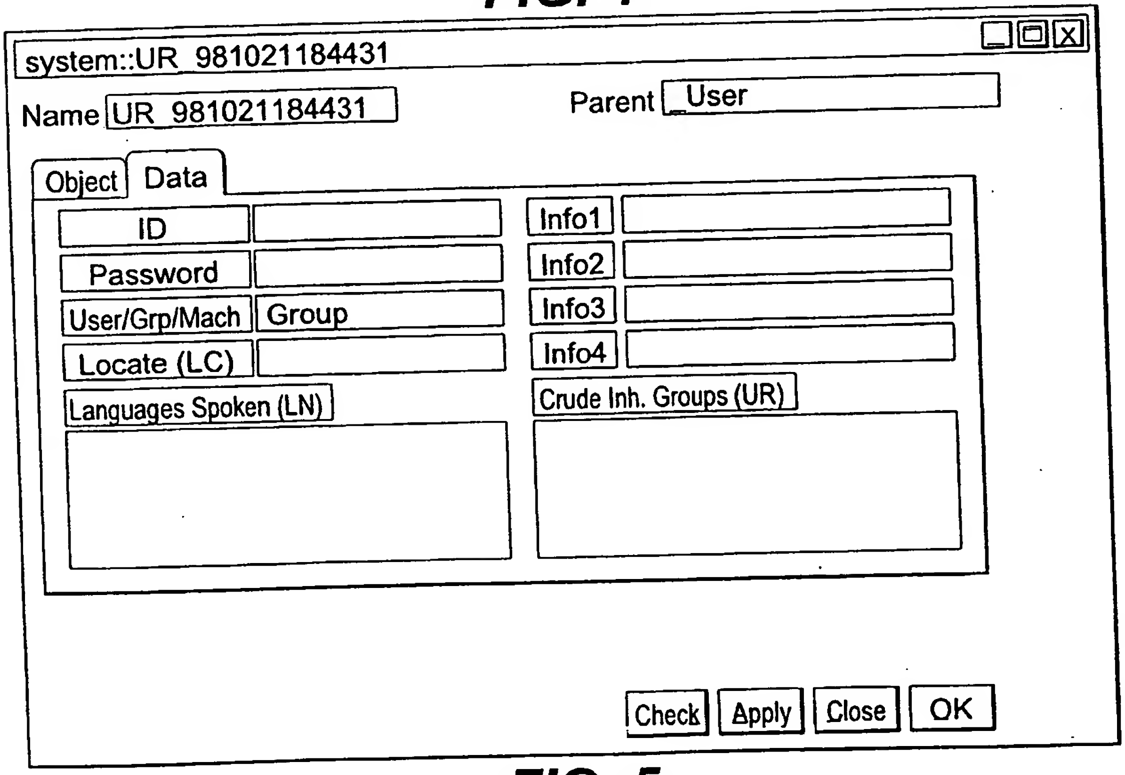
Update		
		Users
1	☺	Admin

Delete		
		User L

Executed		
		All User

□ Cloned

FIG. 4



system::UR_981021184431

Name UR_981021184431

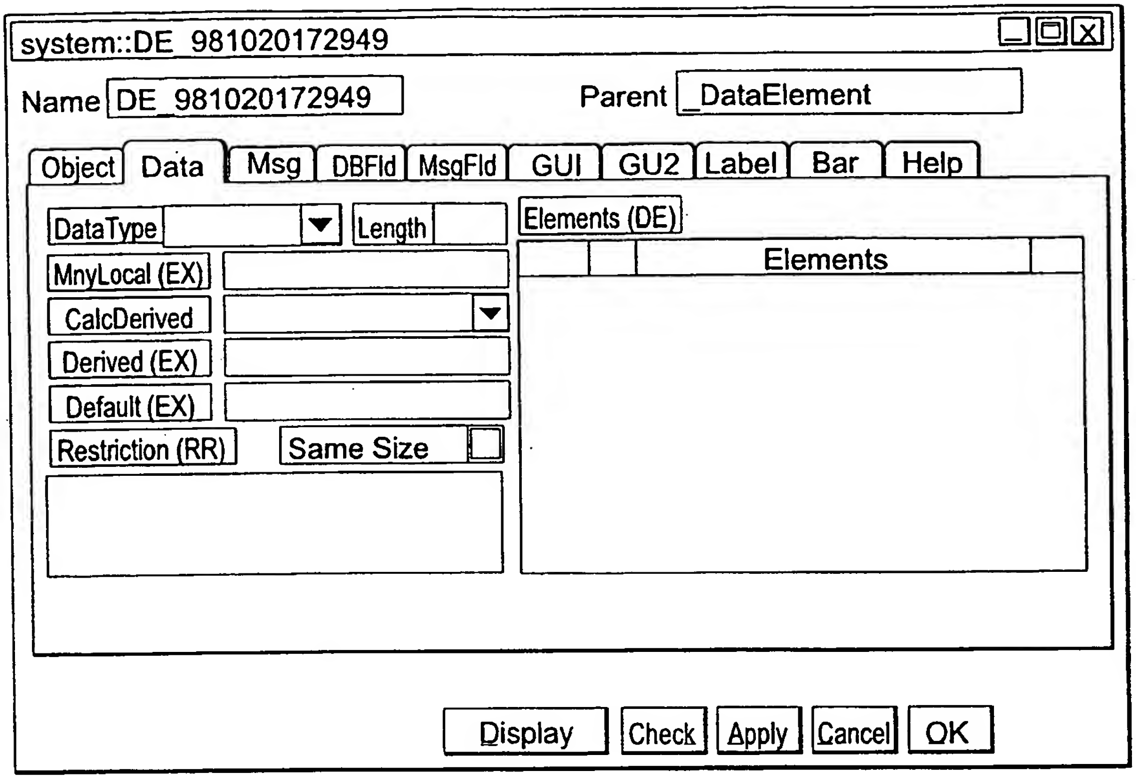
Parent User

ID	
Password	
User/Grp/Mach	Group
Locate (LC)	
Info1	
Info2	
Info3	
Info4	

Languages Spoken (LN)

Crude Inh. Groups (UR)

FIG. 5



system::DE_981020172949

Name: DE_981020172949

Parent: _DataElement

DataType:

Length:

MnyLocal (EX):

CalcDerived:

Derived (EX):

Default (EX):

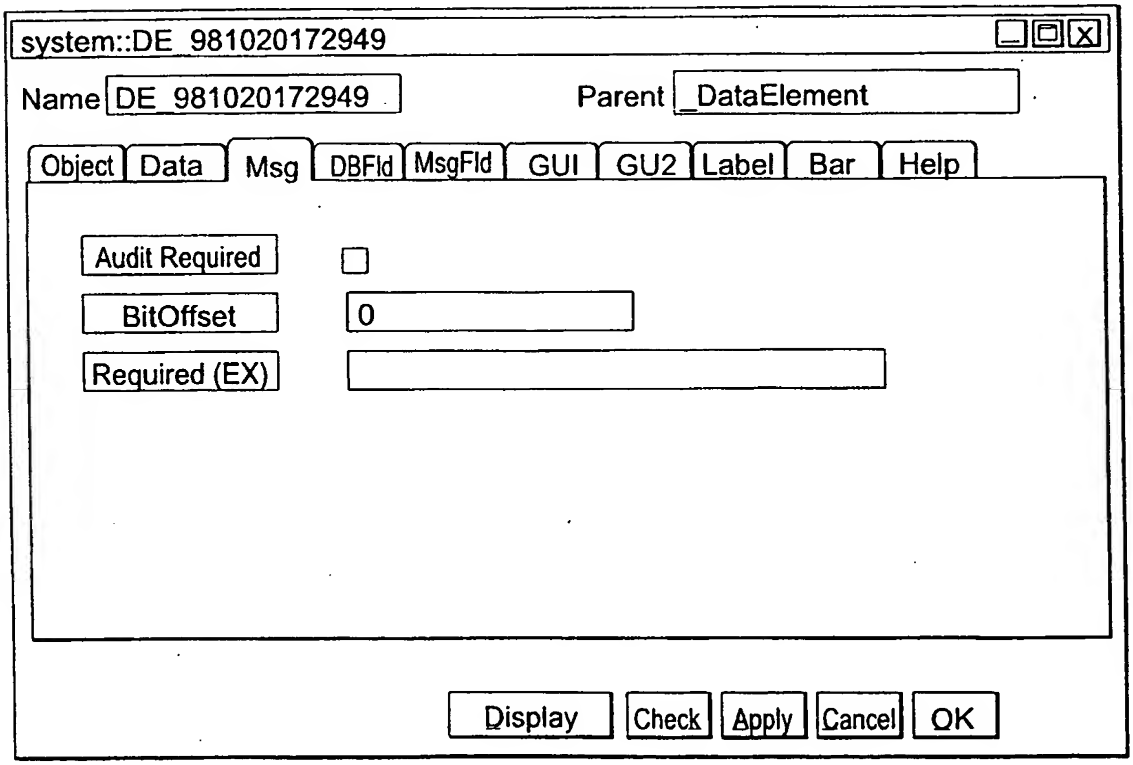
Restriction (RR):

Same Size □

Elements (DE)

		Elements	

FIG. 6A



system::DE_981020172949

Name: DE_981020172949

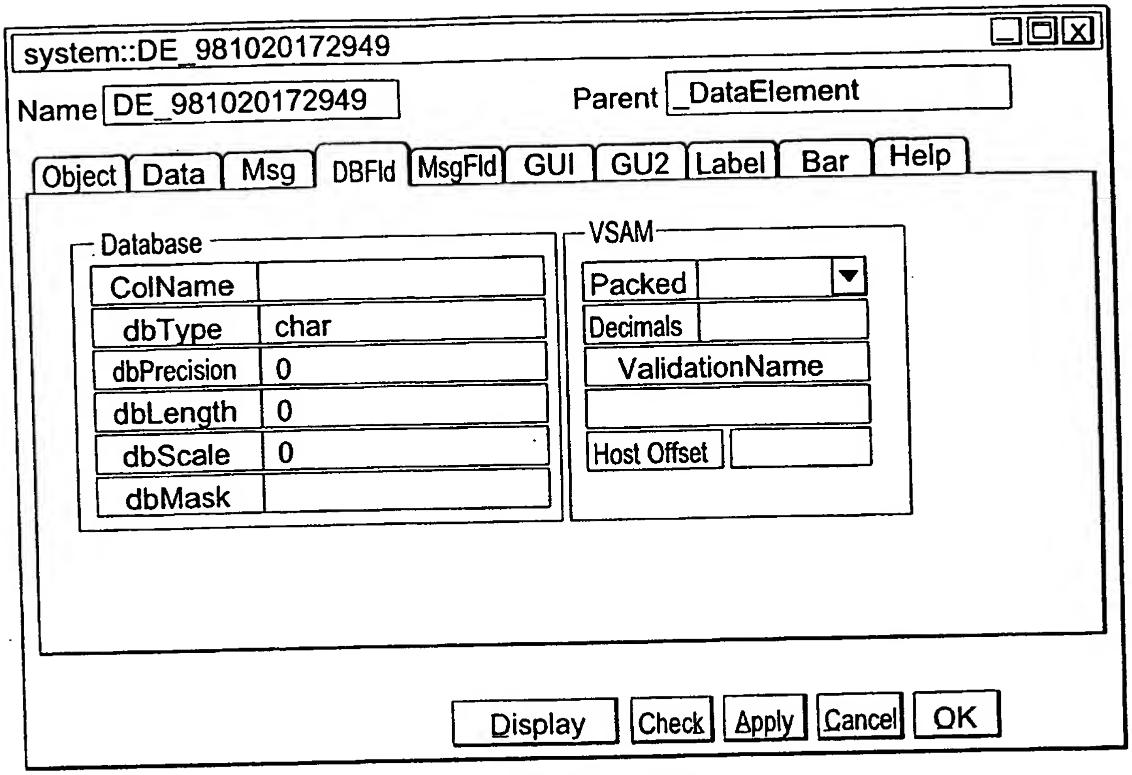
Parent: _DataElement

Audit Required □

BitOffset: 0

Required (EX):

FIG. 6B



system::DE_981020172949

Name DE_981020172949

Parent _DataElement

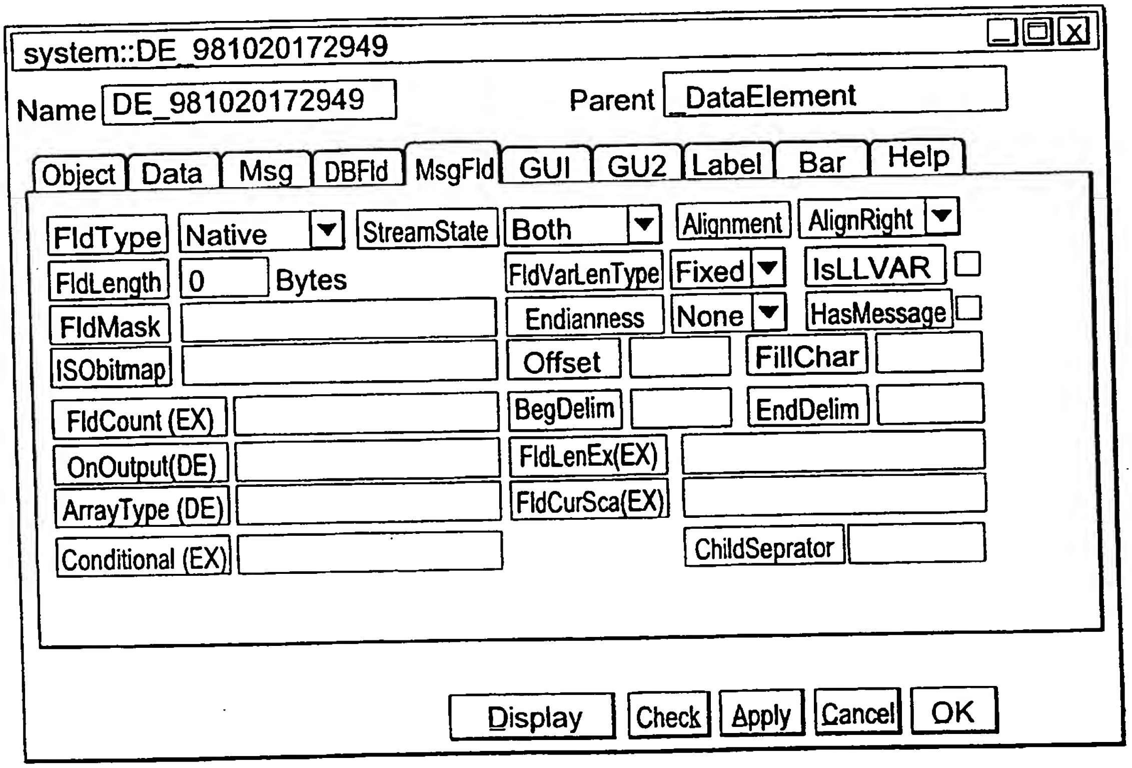
Database

- ColName
- dbType char
- dbPrecision 0
- dbLength 0
- dbScale 0
- dbMask

VSAM

- Packed
- Decimals
- ValidationName
- Host Offset

FIG. 6C



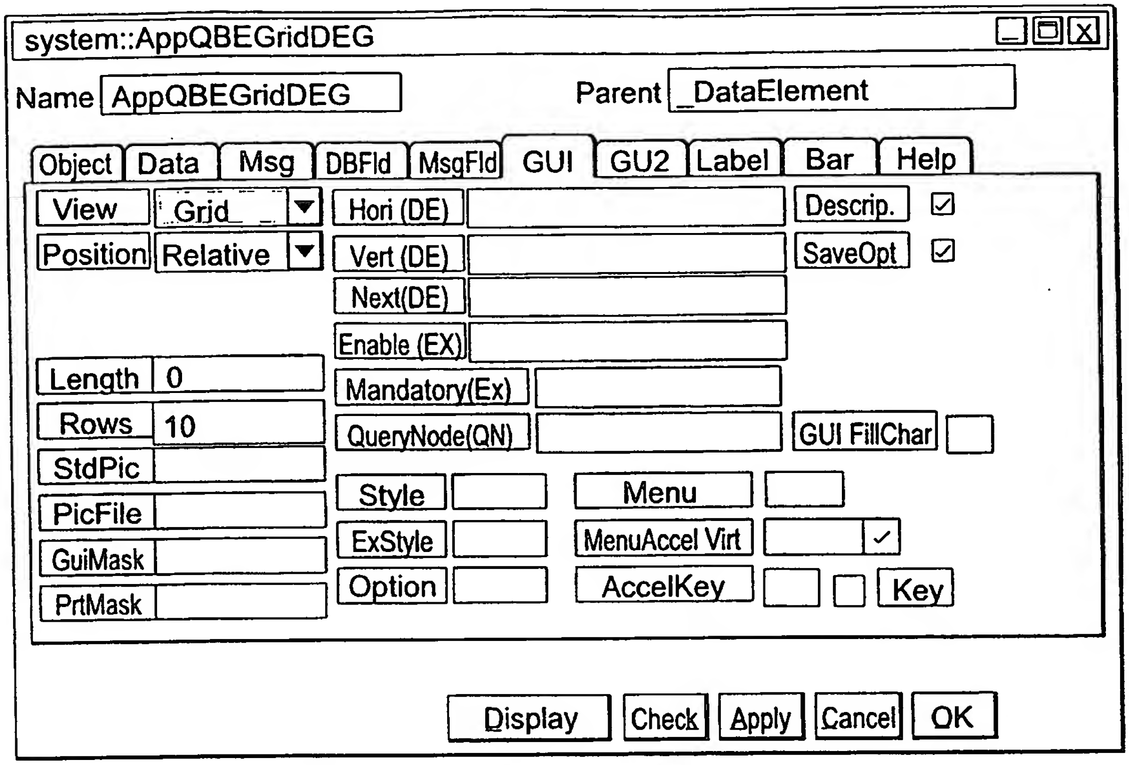
system::DE_981020172949

Name DE_981020172949

Parent _DataElement

- FldType Native
- StreamState Both
- Alignment AlignRight
- FldLength 0 Bytes
- FldVarLenType Fixed
- IsLLVAR □
- FldMask
- Endianness None
- HasMessage □
- ISObitmap
- Offset
- FillChar
- FldCount (EX)
- BegDelim
- EndDelim
- OnOutput(DE)
- FldLenEx(EX)
- ArrayType (DE)
- FldCurSca(EX)
- Conditional (EX)
- ChildSeprator

FIG. 6D



system::AppQBEGridDEG

Name AppQBEGridDEG

Parent _DataElement

View Grid

Position Relative

Length 0

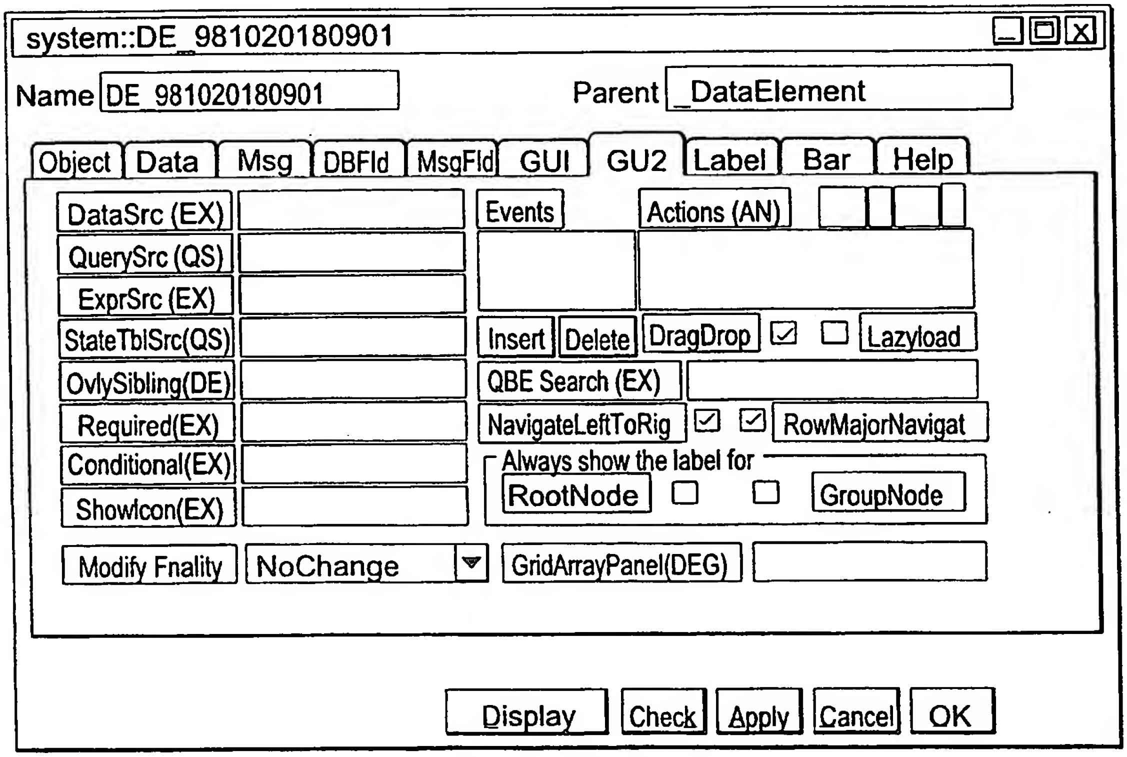
Rows 10

Descrip. ☑

SaveOpt ☑

MenuAccel Virt ✓

FIG. 6E



system::DE_981020180901

Name DE_981020180901

Parent _DataElement

DragDrop ☑

□ Lazyload

NavigateLeftToRig ☑

☑ RowMajorNavigat

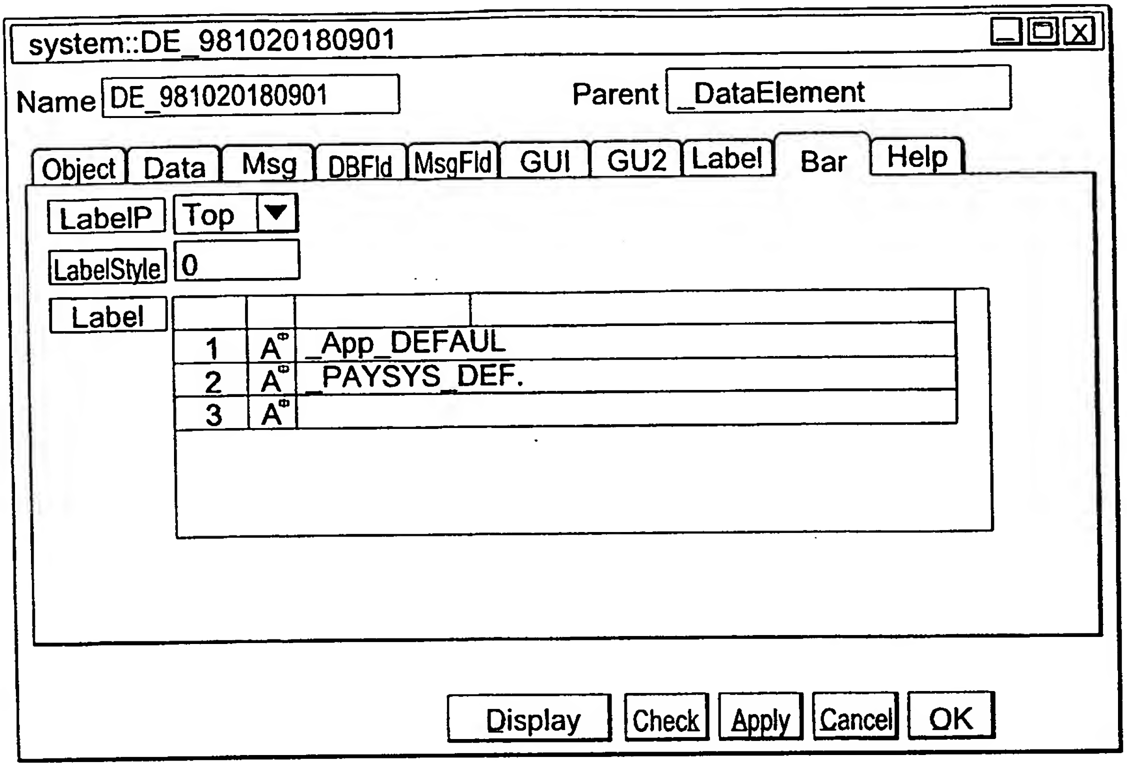
Always show the label for

RootNode □

□ GroupNode

Modify Fnality NoChange

FIG. 6F



system::DE_981020180901

Name DE_981020180901 Parent _DataElement

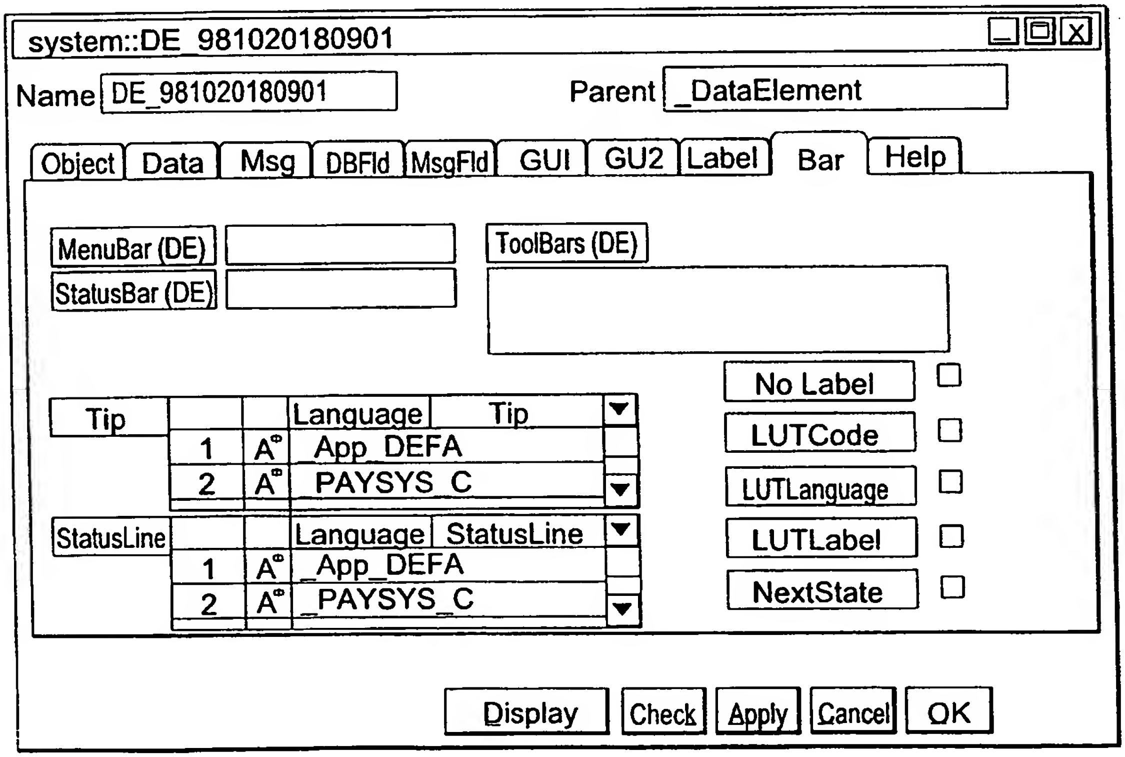
LabelP: Top ▼

LabelStyle: 0

Label

1	A	_App_DEFAUL
2	A	_PAYSYS_DEF.
3	A	

FIG. 6G



system::DE_981020180901

Name DE_981020180901 Parent _DataElement

MenuBar (DE)

StatusBar (DE)

ToolBars (DE)

Tip

		Language	Tip
1	A	_App_DEFA	
2	A	_PAYSYS_C	

StatusLine

		Language	StatusLine
1	A	_App_DEFA	
2	A	_PAYSYS_C	

No Label □

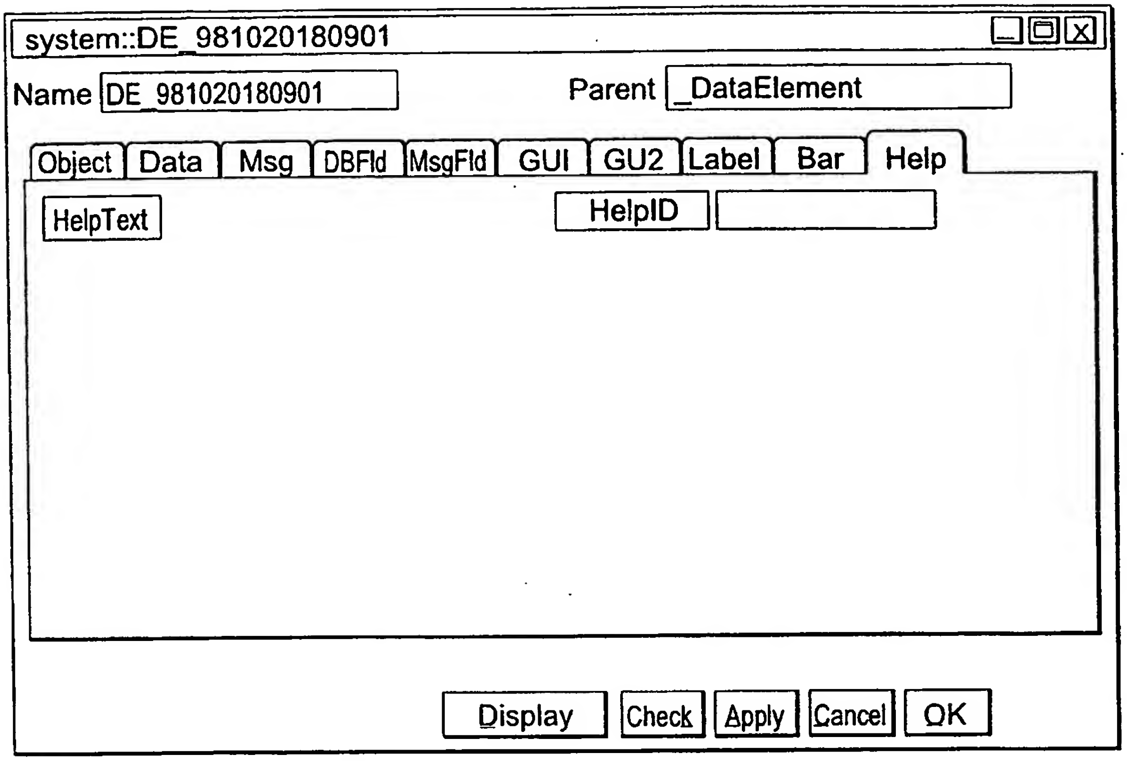
LUTCode □

LUTLanguage □

LUTLabel □

NextState □

FIG. 6H



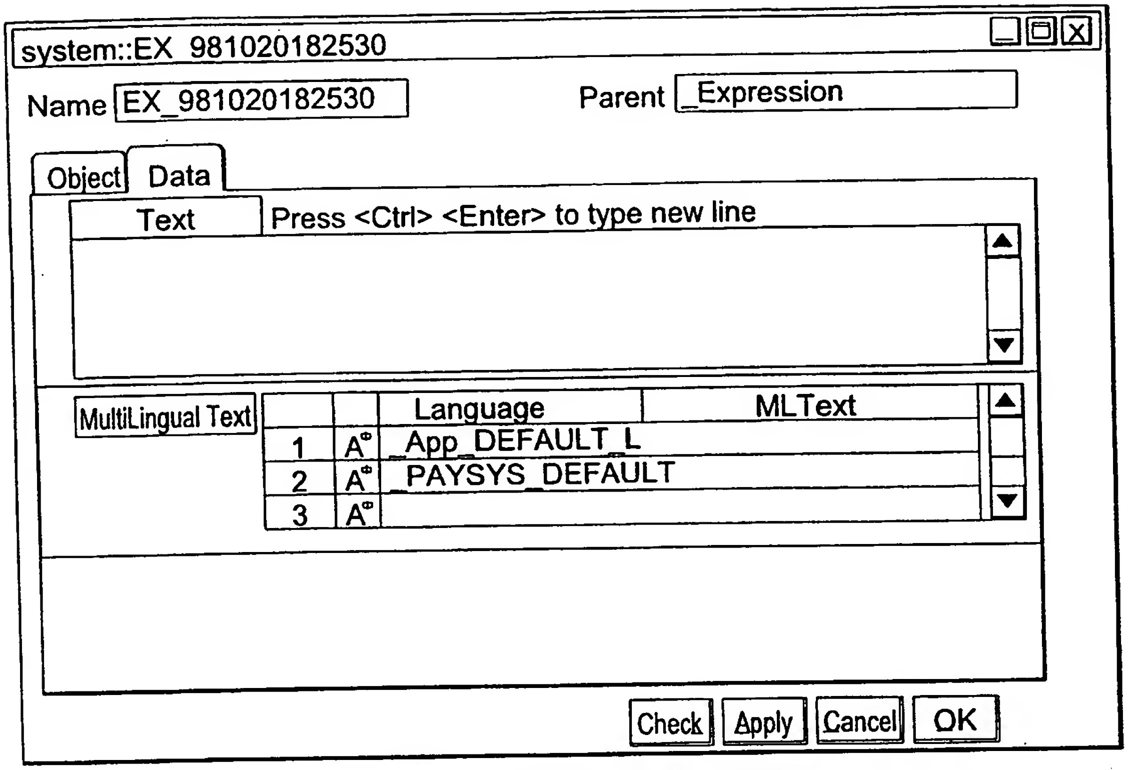
system::DE_981020180901

Name DE_981020180901

Parent _DataElement

HelpID

FIG. 6I



system::EX_981020182530

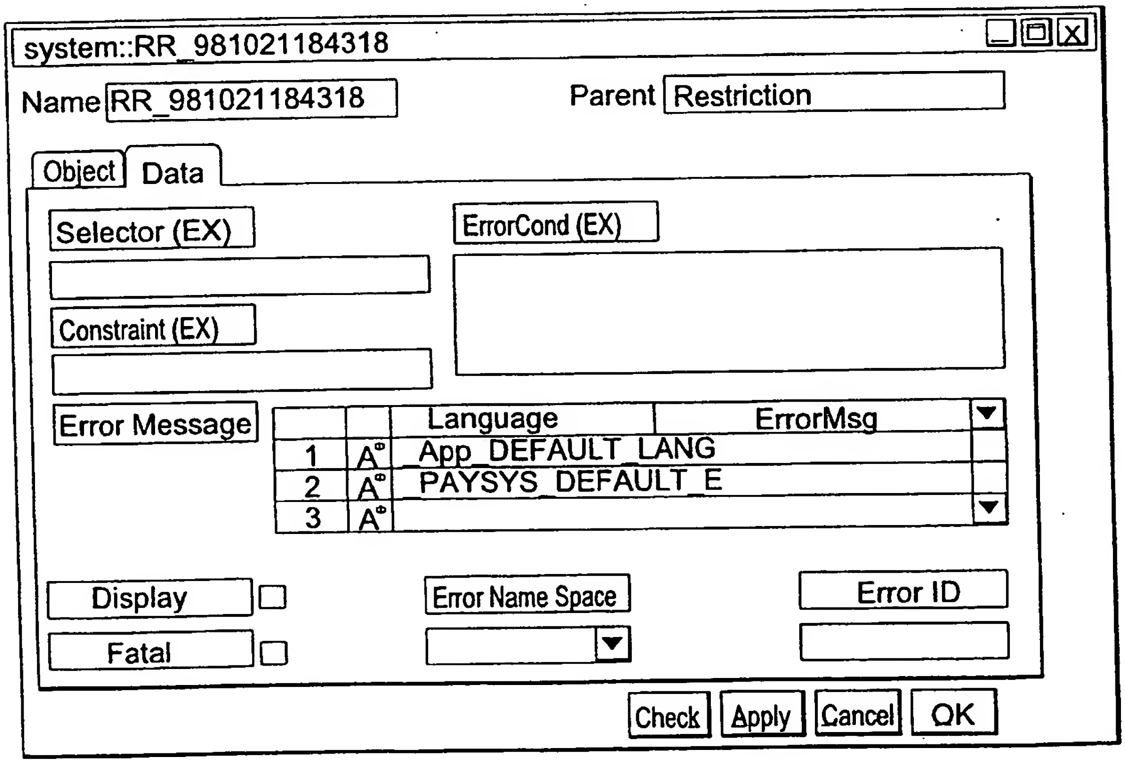
Name EX_981020182530 Parent _Expression

Text Press <Ctrl> <Enter> to type new line

MultiLingual Text

		Language	MLText
1	A	_App_DEFAULT_L	
2	A	_PAYSYS_DEFAULT	
3	A		

FIG. 7



system::RR_981021184318

Name RR_981021184318 Parent Restriction

Selector (EX)

Constraint (EX)

ErrorCond (EX)

Error Message

		Language	ErrorMsg
1	A	_App_DEFAULT_LANG	
2	A	_PAYSYS_DEFAULT_E	
3	A		

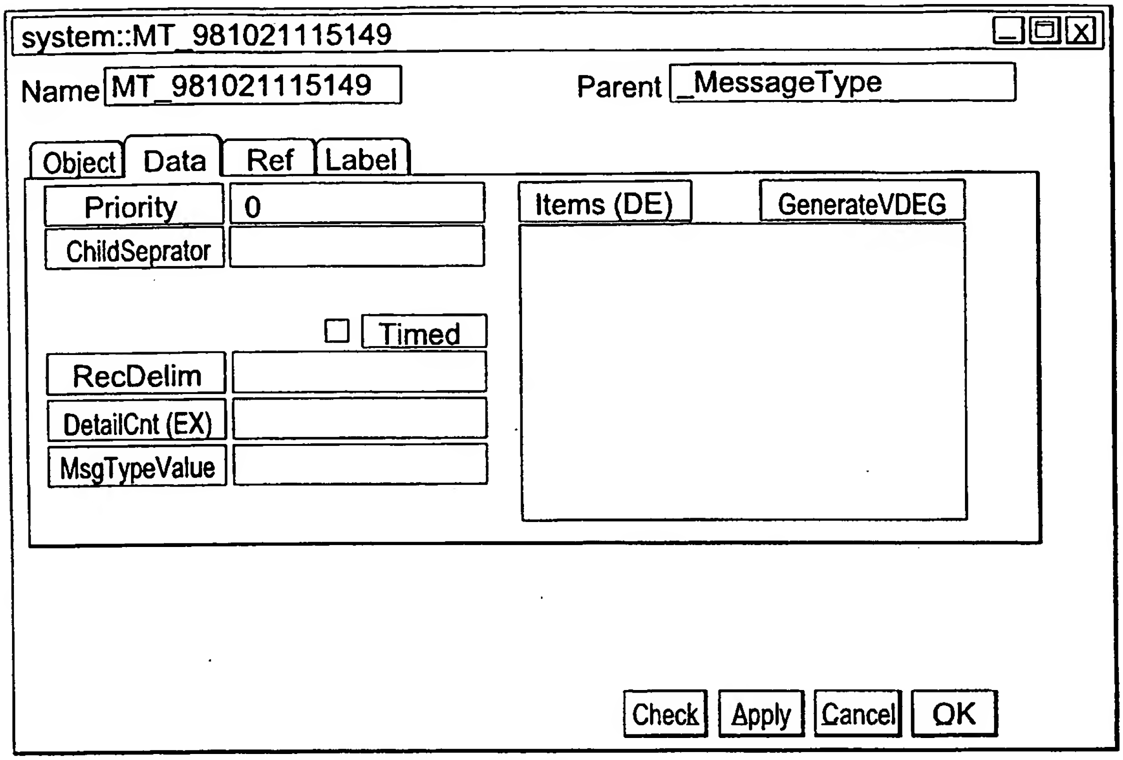
Display □

Fatal □

Error Name Space

Error ID

FIG. 8



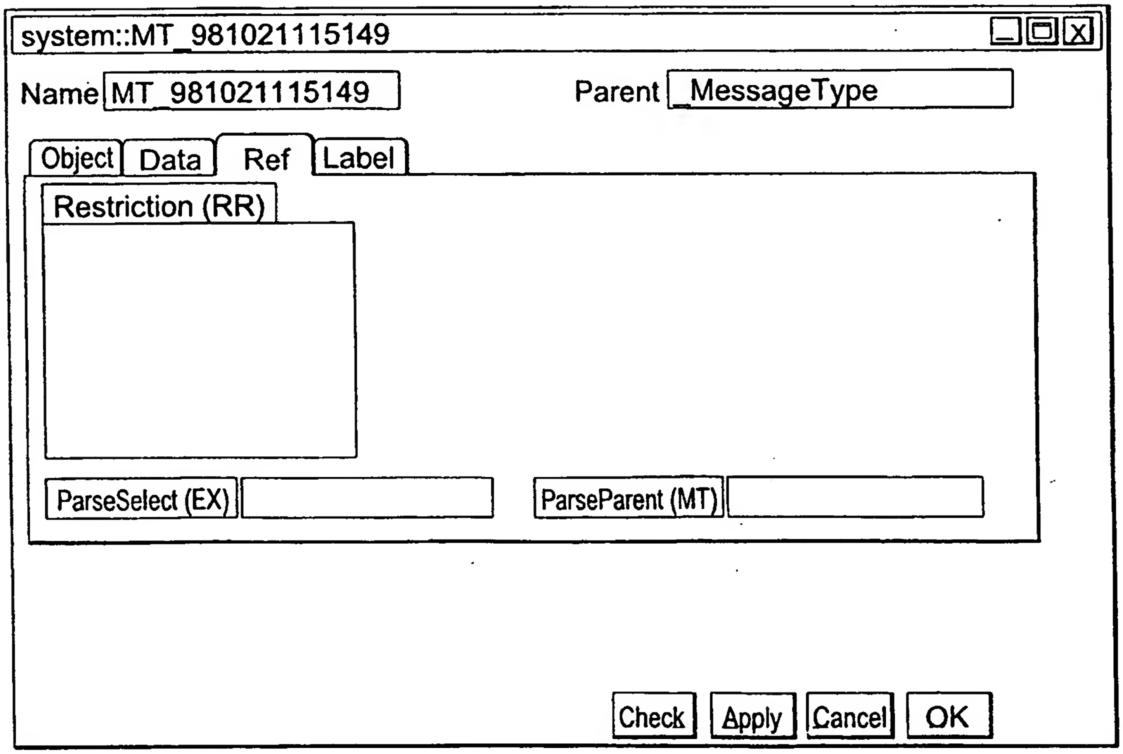
system::MT_981021115149

Name MT_981021115149

Parent _MessageType

Priority 0

FIG. 9A

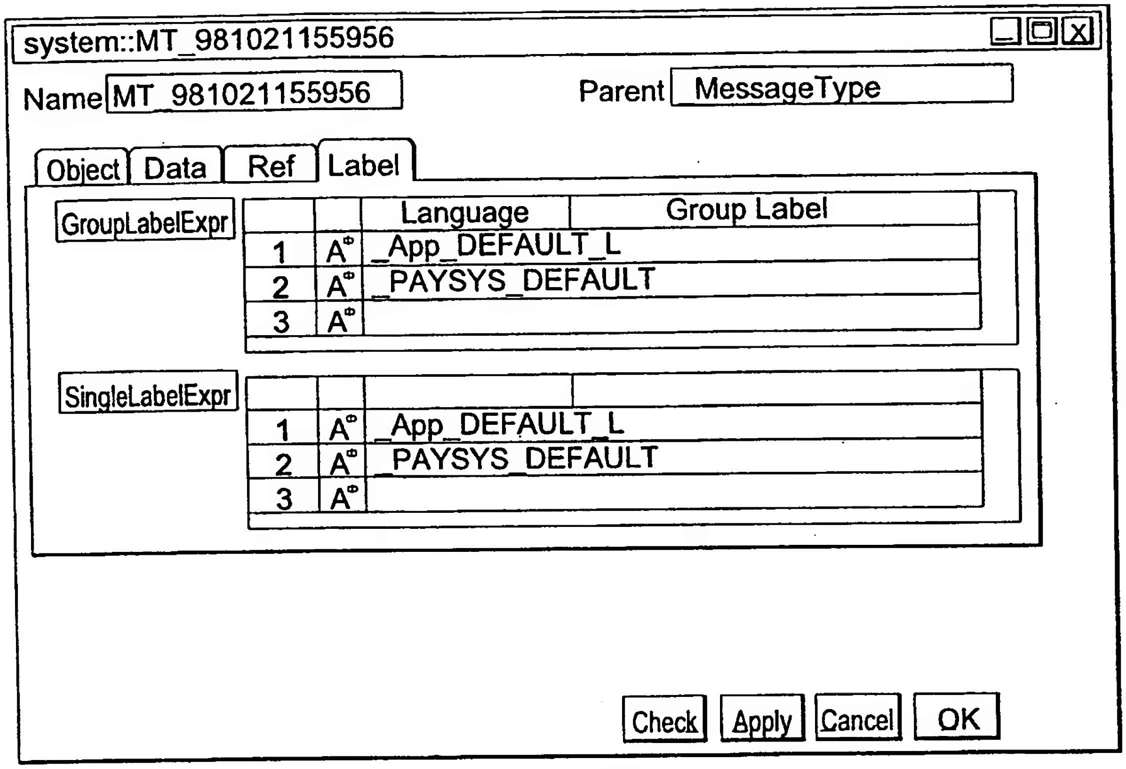


system::MT_981021115149

Name MT_981021115149

Parent _MessageType

FIG. 9B



system::MT_981021155956

Name MT_981021155956

Parent MessageType

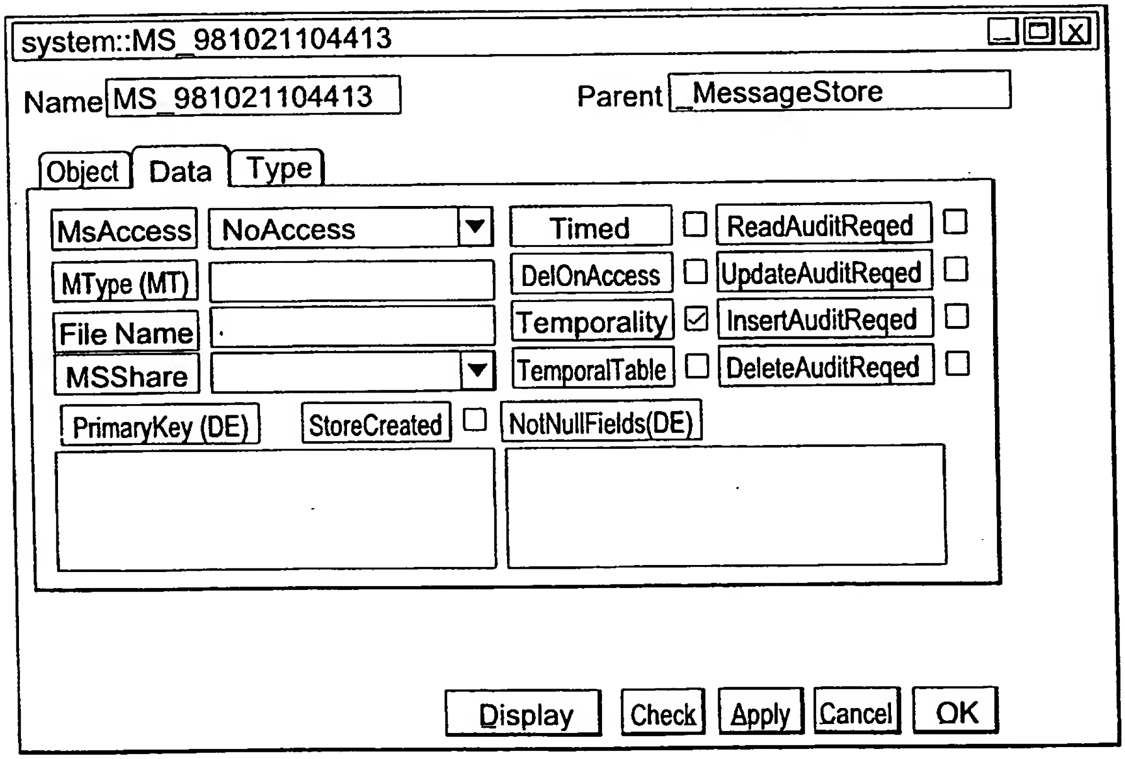
GroupLabelExpr

		Language	Group Label
1	A°	_App_DEFAULT_L	
2	A°	_PAYSYS_DEFAULT	
3	A°		

SingleLabelExpr

1	A°	_App_DEFAULT_L	
2	A°	_PAYSYS_DEFAULT	
3	A°		

FIG. 9C



system::MS_981021104413

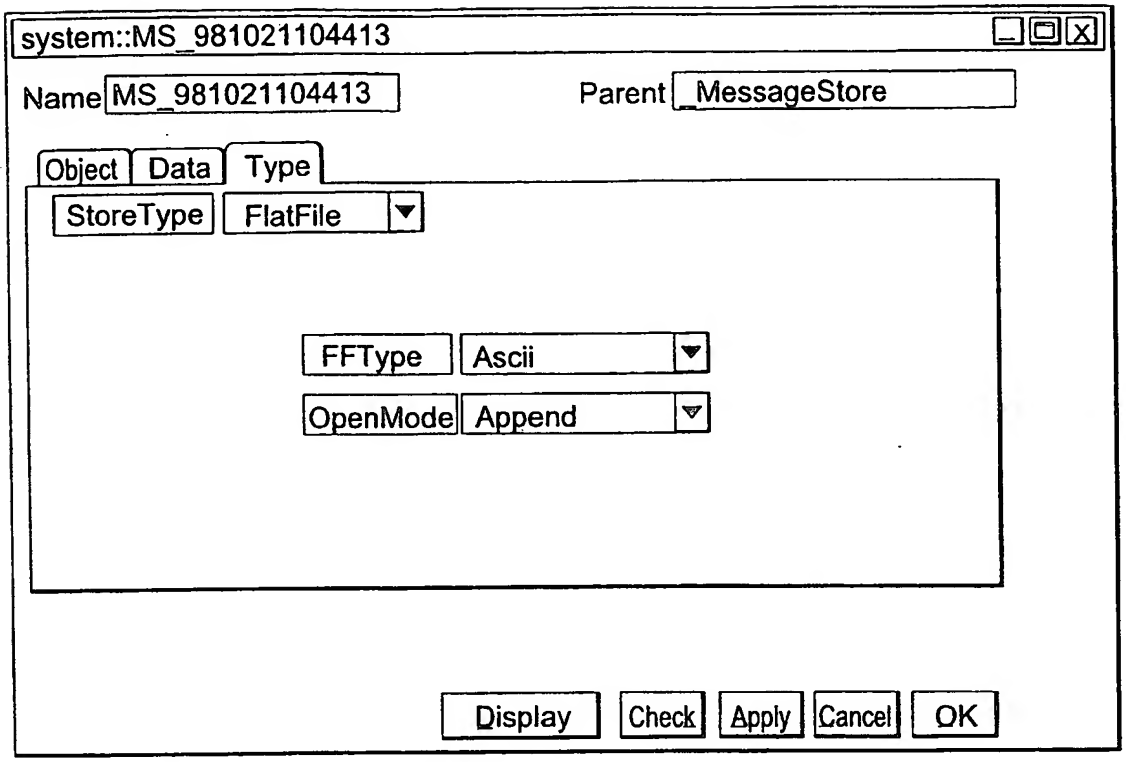
Name MS_981021104413

Parent _MessageStore

MsAccess	NoAccess ▼	Timed	□	ReadAuditReqed	□
MType (MT)		DelOnAccess	□	UpdateAuditReqed	□
File Name		Temporality	☑	InsertAuditReqed	□
MSShare	▼	TemporalTable	□	DeleteAuditReqed	□

StoreCreated □

FIG. 10A



system::MS_981021104413

Name MS_981021104413

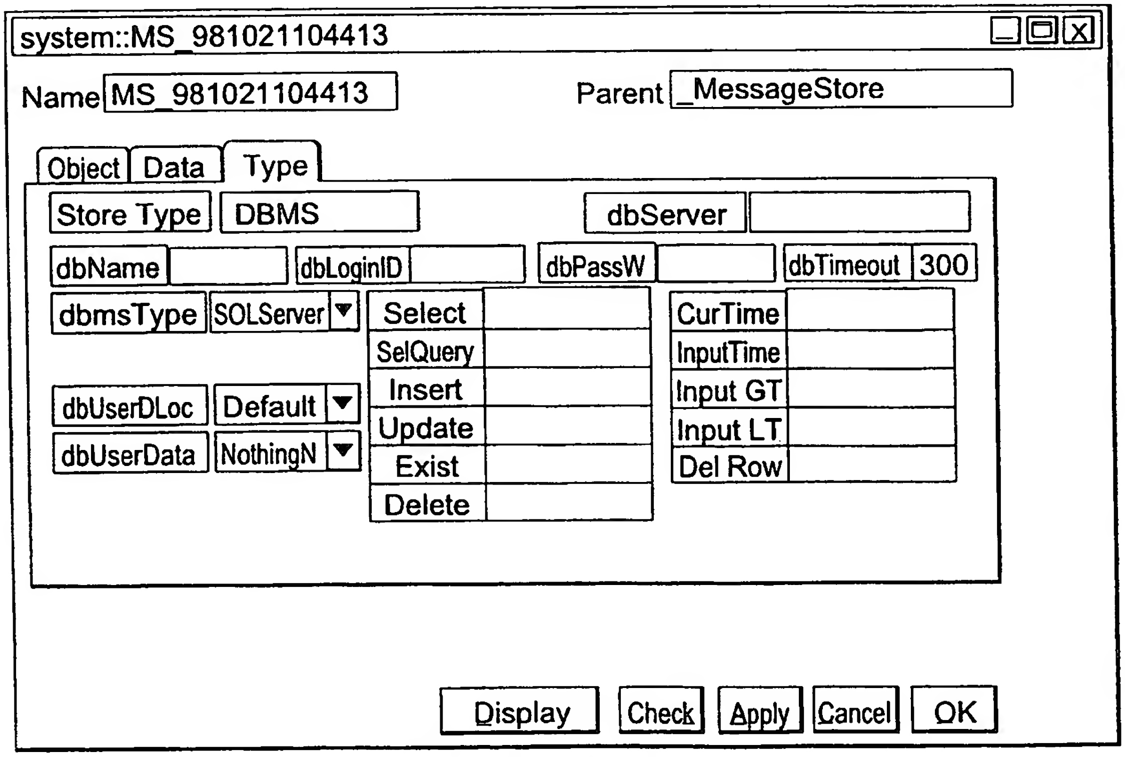
Parent _MessageStore

StoreType FlatFile

FFType Ascii

OpenMode Append

FIG. 10B



system::MS_981021104413

Name MS_981021104413

Parent _MessageStore

Store Type DBMS

dbServer

dbName

dbLoginID

dbPassW

dbTimeout 300

dbmsType SOLServer

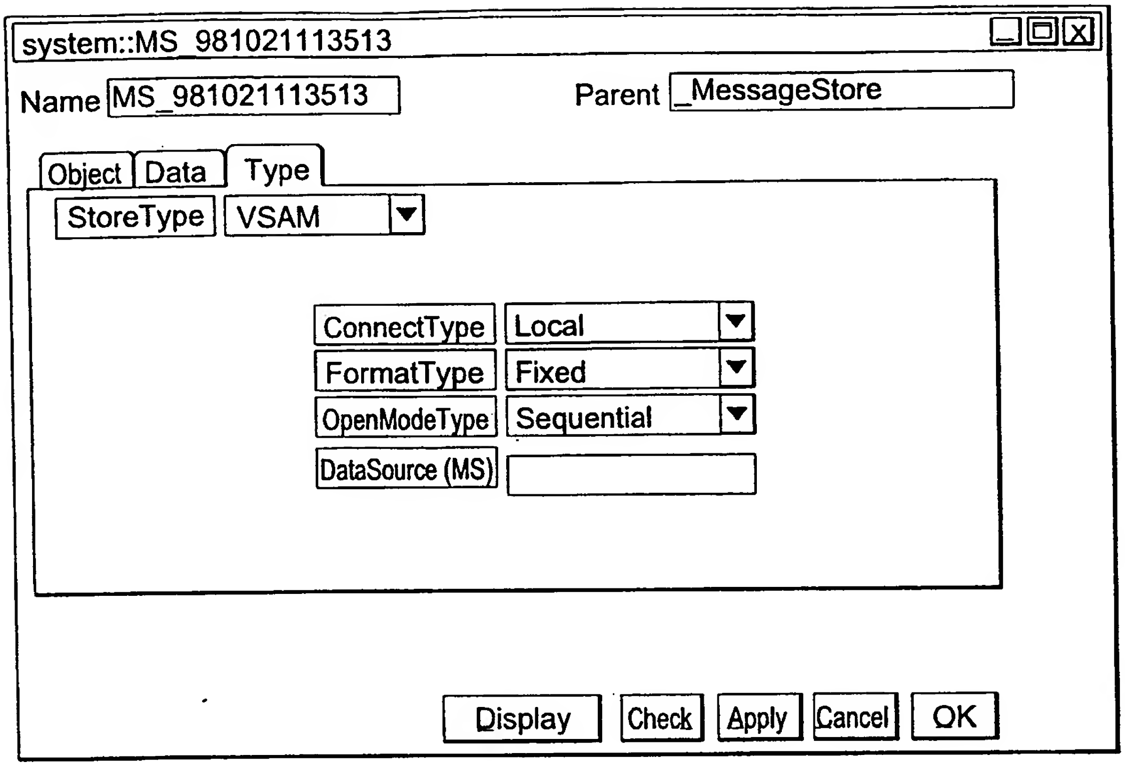
dbUserDLoc Default

dbUserData NothingN

Select	
SelQuery	
Insert	
Update	
Exist	
Delete	

CurTime	
InputTime	
Input GT	
Input LT	
Del Row	

FIG. 10C



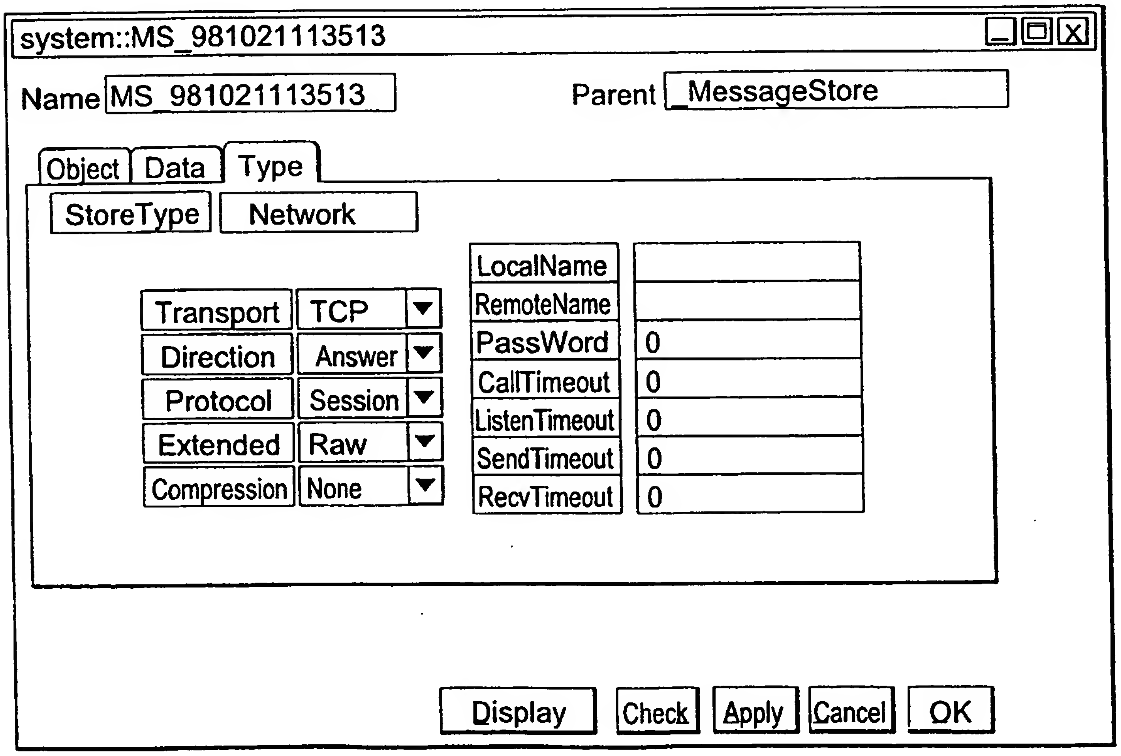
system::MS_981021113513

Name MS_981021113513

Parent _MessageStore

StoreType	VSAM
ConnectType	Local
FormatType	Fixed
OpenModeType	Sequential
DataSource (MS)	

FIG. 10D



system::MS_981021113513

Name MS_981021113513

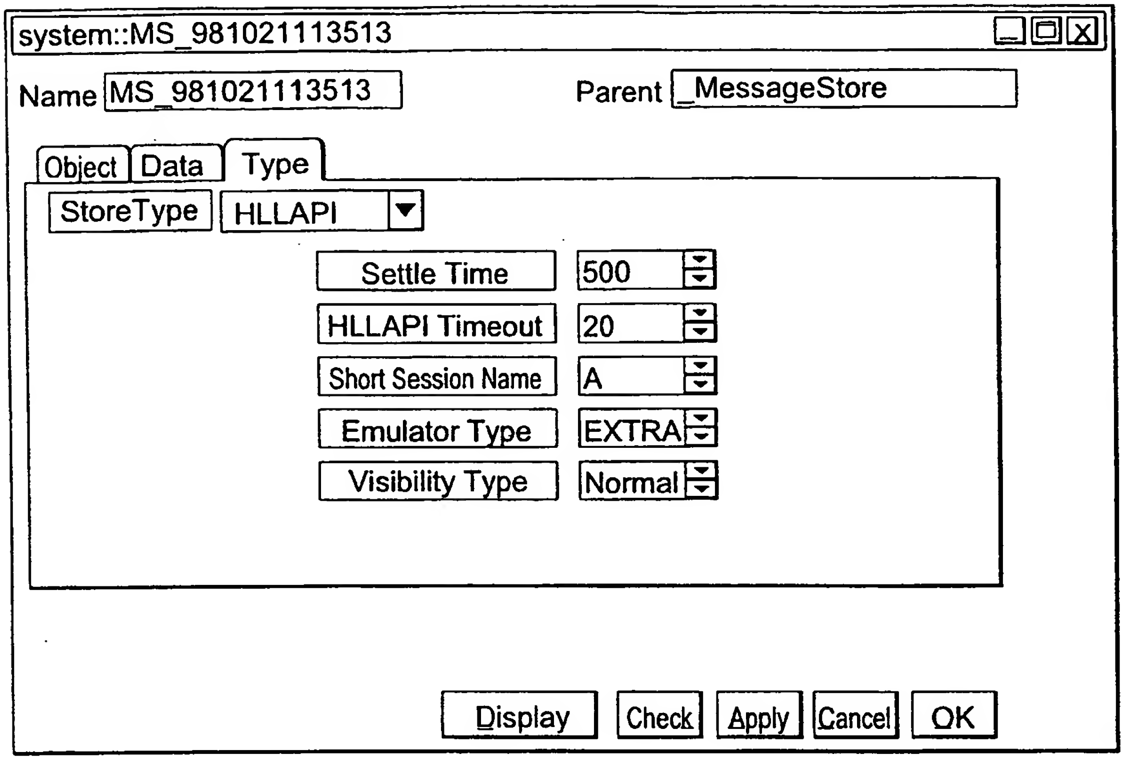
Parent _MessageStore

StoreType	Network

Transport	TCP
Direction	Answer
Protocol	Session
Extended	Raw
Compression	None

LocalName	
RemoteName	
PassWord	0
CallTimeout	0
ListenTimeout	0
SendTimeout	0
RecvTimeout	0

FIG. 10E



system::MS_981021113513

Name MS_981021113513

Parent _MessageStore

StoreType HLLAPI

Settle Time 500

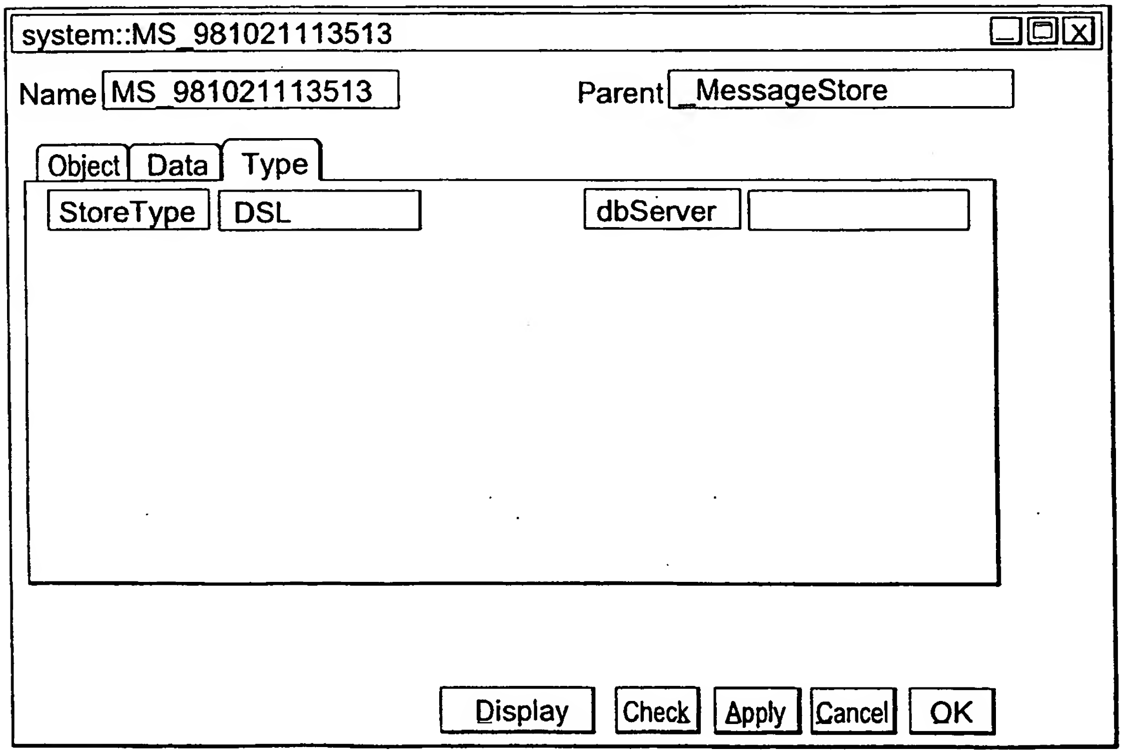
HLLAPI Timeout 20

Short Session Name A

Emulator Type EXTRA

Visibility Type Normal

FIG. 10F



system::MS_981021113513

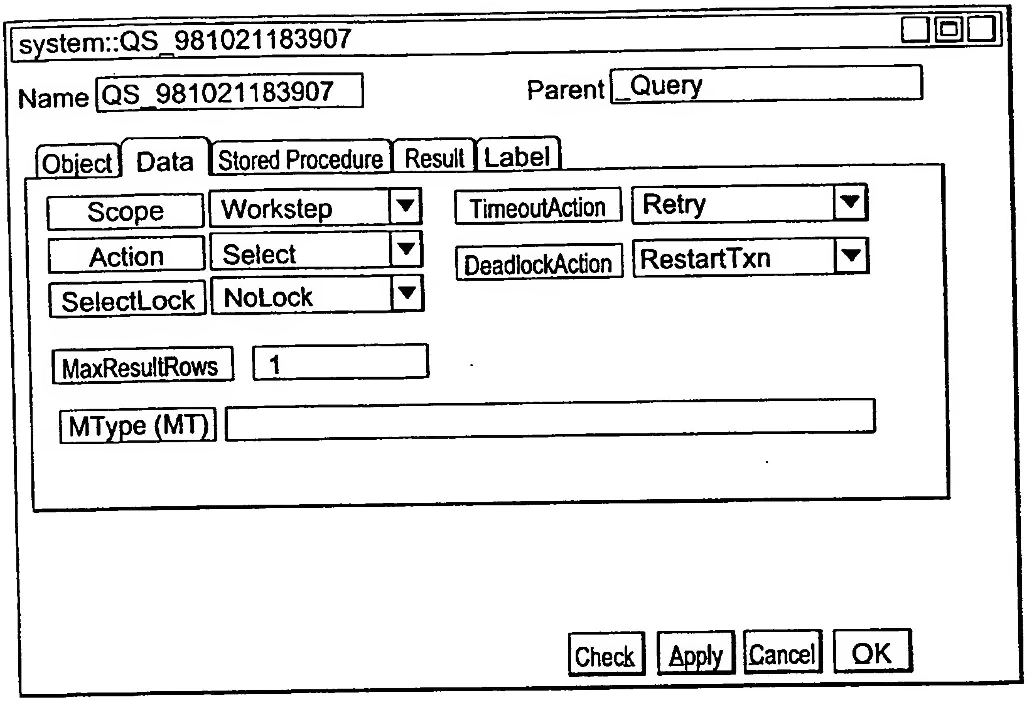
Name MS_981021113513

Parent _MessageStore

StoreType DSL

dbServer

FIG. 10G


system::QS_981021183907

Name QS_981021183907

Parent _Query

Scope Workstep

Action Select

SelectLock NoLock

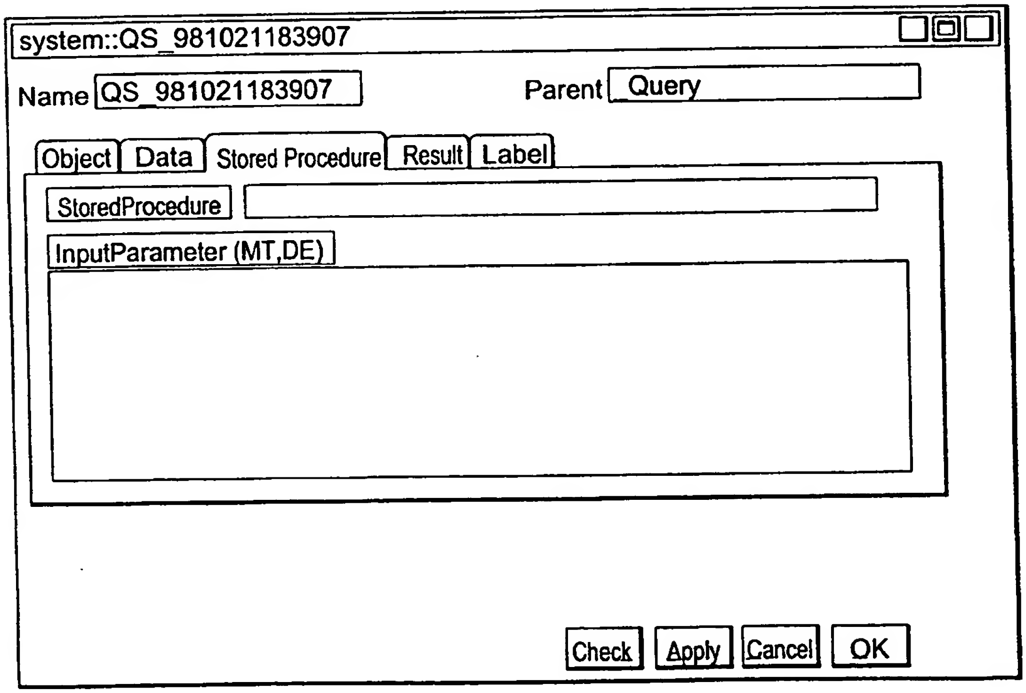
TimeoutAction Retry

DeadlockAction RestartTxn

MaxResultRows 1

MType (MT)

FIG. 11A


system::QS_981021183907

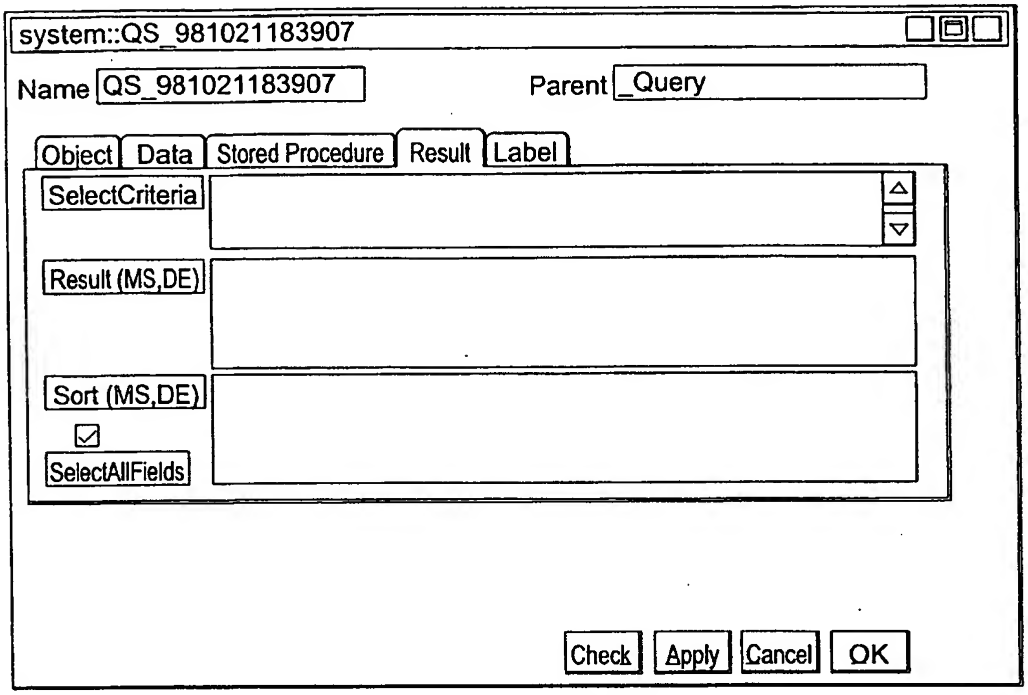
Name QS_981021183907

Parent Query

StoredProcedure

InputParameter (MT,DE)

FIG. 11B

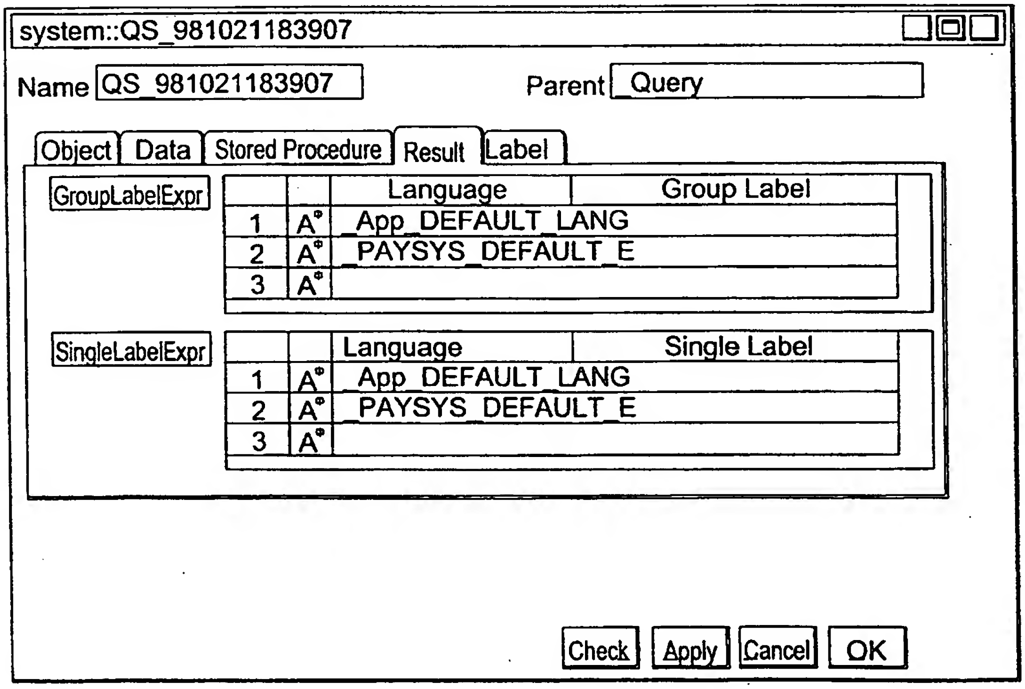


system::QS_981021183907

Name QS_981021183907

Parent _Query

FIG. 11C



system::QS_981021183907

Name QS_981021183907

Parent _Query

		Language	Group Label
1	A*	_App_DEFAULT_LANG	
2	A*	_PAYSYS_DEFAULT_E	
3	A*		

		Language	Single Label
1	A*	_App_DEFAULT_LANG	
2	A*	_PAYSYS_DEFAULT_E	
3	A*		

FIG. 11D

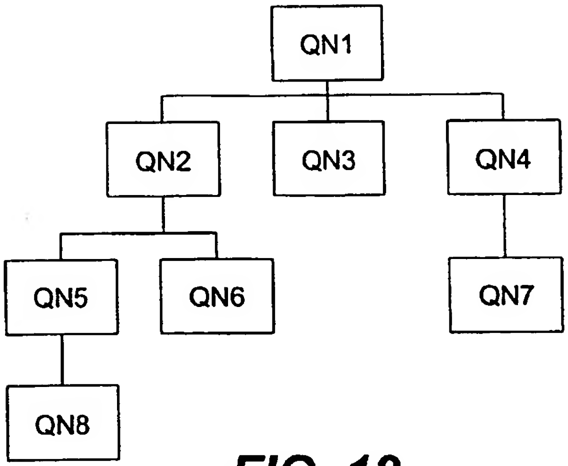
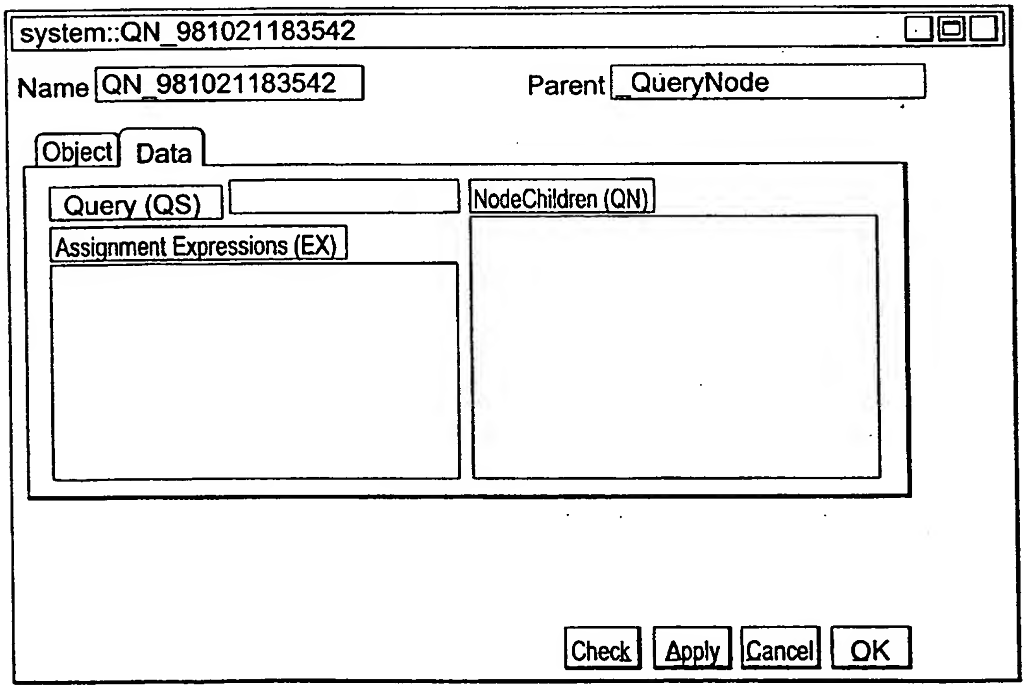


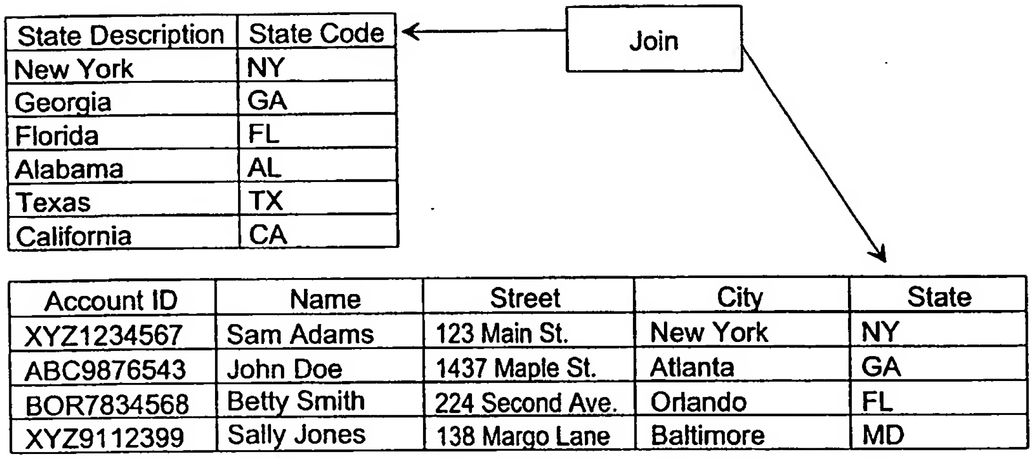
FIG. 12


system::QN_981021183542

Name QN_981021183542

Parent QueryNode

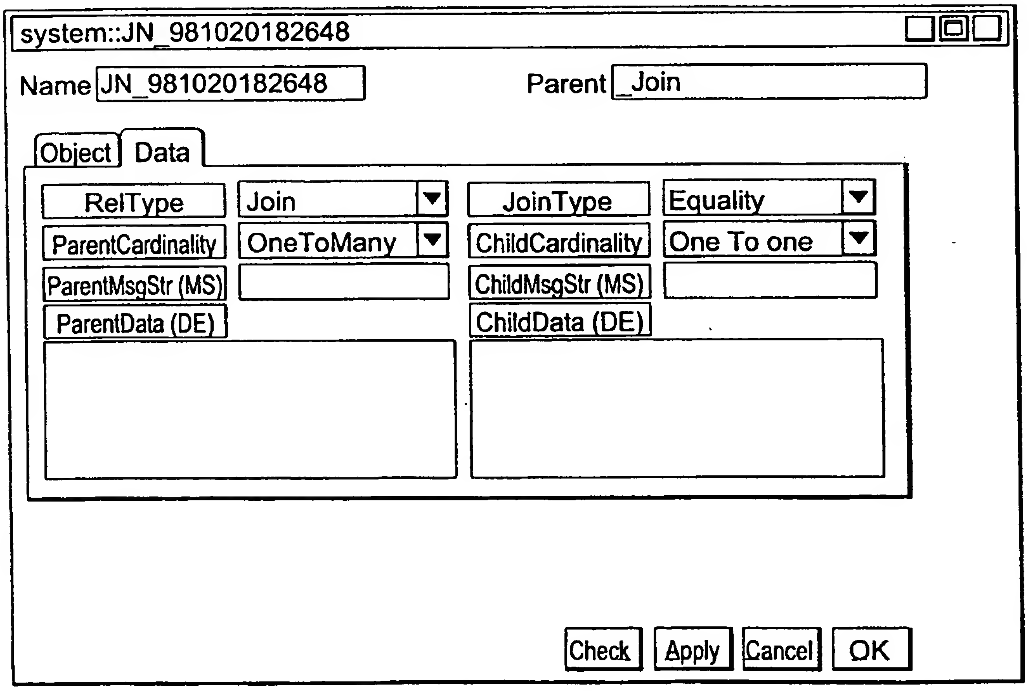
FIG. 13



State Description	State Code
New York	NY
Georgia	GA
Florida	FL
Alabama	AL
Texas	TX
California	CA

Account ID	Name	Street	City	State
XYZ1234567	Sam Adams	123 Main St.	New York	NY
ABC9876543	John Doe	1437 Maple St.	Atlanta	GA
BOR7834568	Betty Smith	224 Second Ave.	Orlando	FL
XYZ9112399	Sally Jones	138 Margo Lane	Baltimore	MD

FIG. 14



system::JN_981020182648

Name JN_981020182648

Parent _Join

RelType	Join	JoinType	Equality
ParentCardinality	OneToMany	ChildCardinality	One To one
ParentMsgStr (MS)		ChildMsgStr (MS)	
ParentData (DE)		ChildData (DE)	

FIG. 15

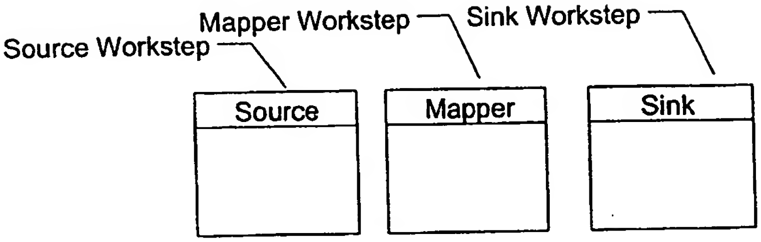
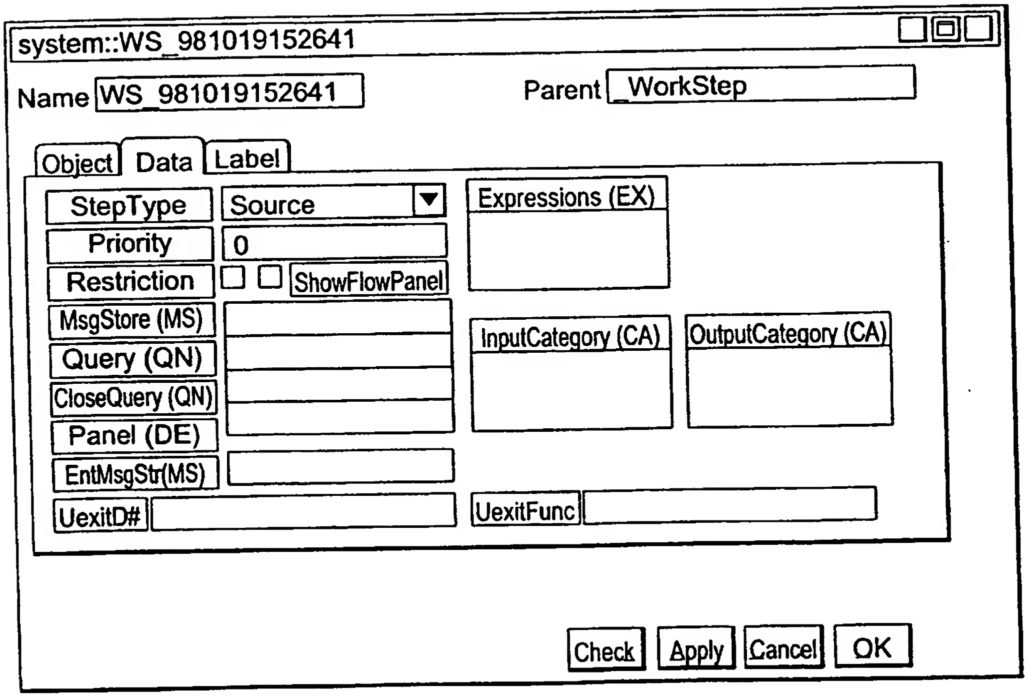


FIG. 16


system::WS_981019152641

Name WS_981019152641

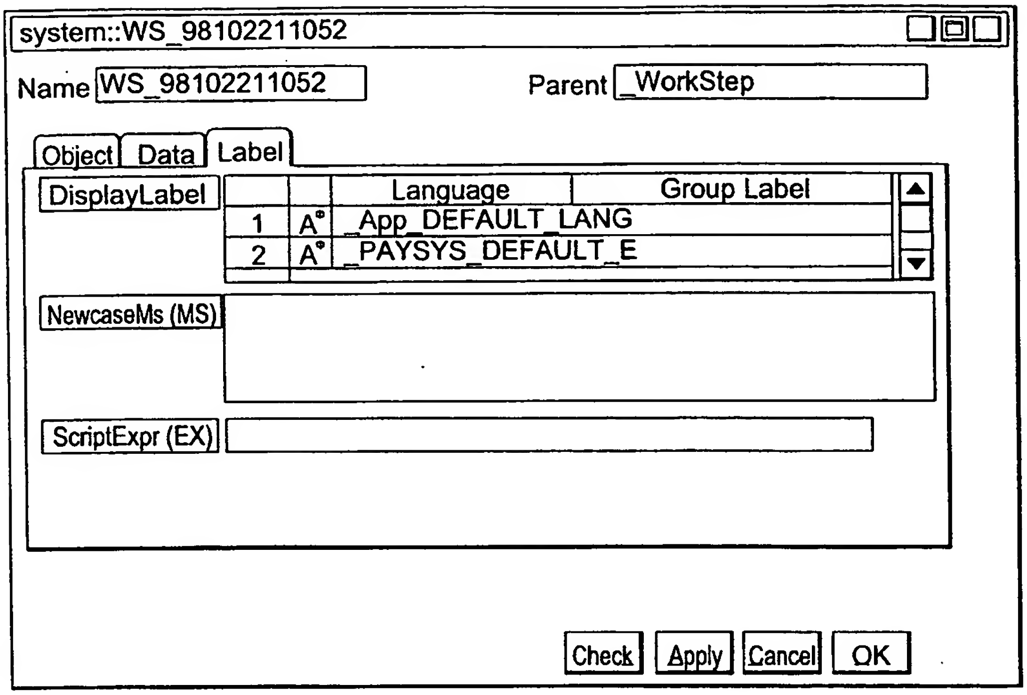
Parent _WorkStep

StepType: Source

Priority: 0

Restriction: □ □ ShowFlowPanel

FIG. 17A



system::WS_98102211052

Name WS_98102211052

Parent _WorkStep

DisplayLabel

		Language	Group Label
1	A°	_App_DEFAULT_LANG	
2	A°	_PAYSYS_DEFAULT_E	

NewcaseMs (MS)

ScriptExpr (EX)

FIG. 17B

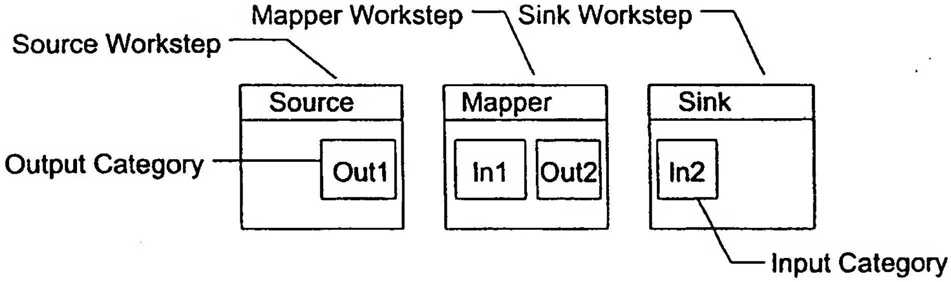
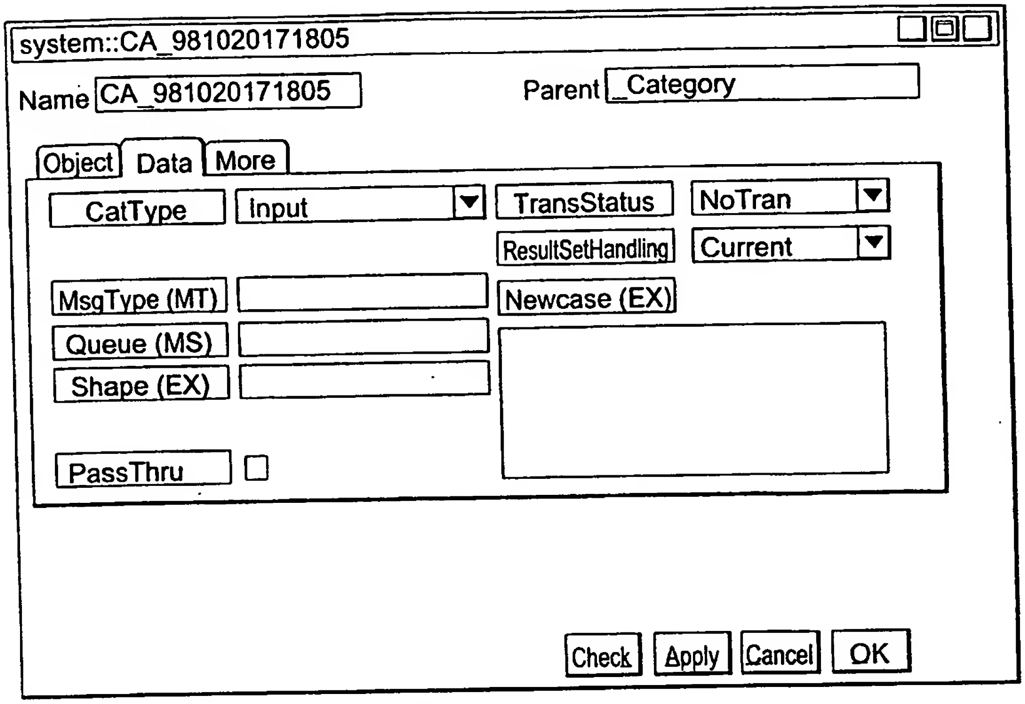


FIG. 18


system::CA_981020171805

Name CA_981020171805

Parent _Category

CatType Input

TransStatus NoTran

ResultSetHandling Current

MsgType (MT)

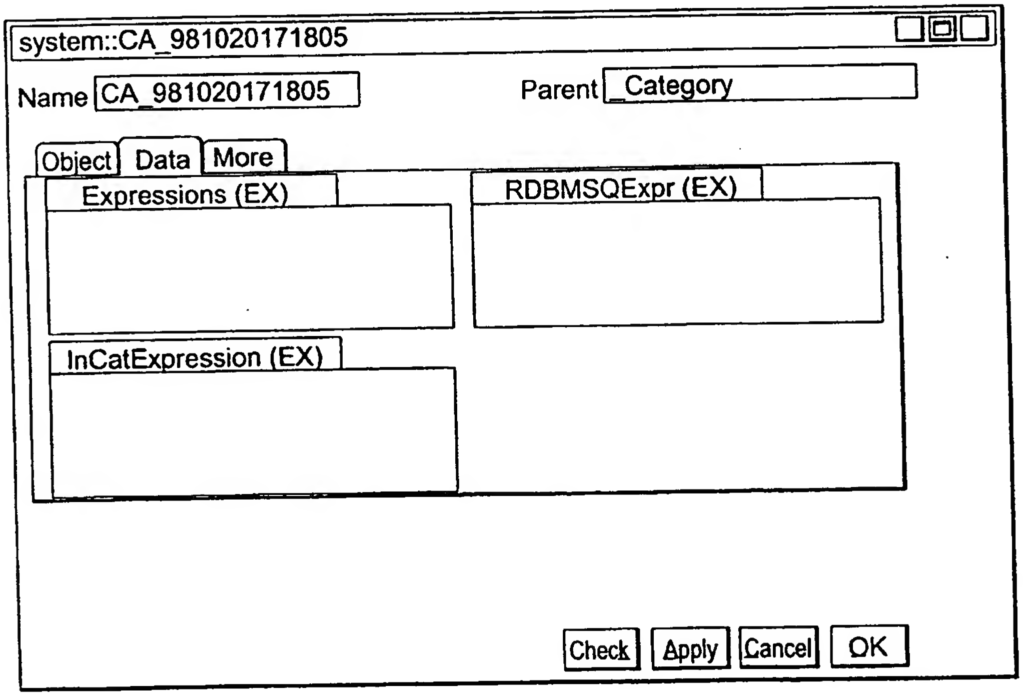
Queue (MS)

Shape (EX)

Newcase (EX)

PassThru □

FIG. 19A


system::CA_981020171805

Name CA_981020171805

Parent _Category

Expressions (EX)

RDBMSQExpr (EX)

InCatExpression (EX)

FIG. 19B

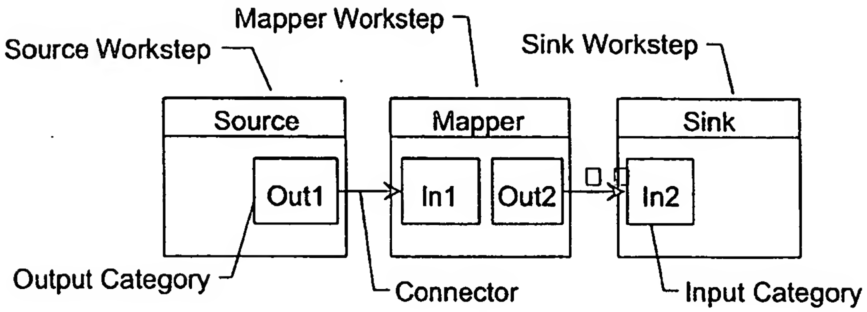
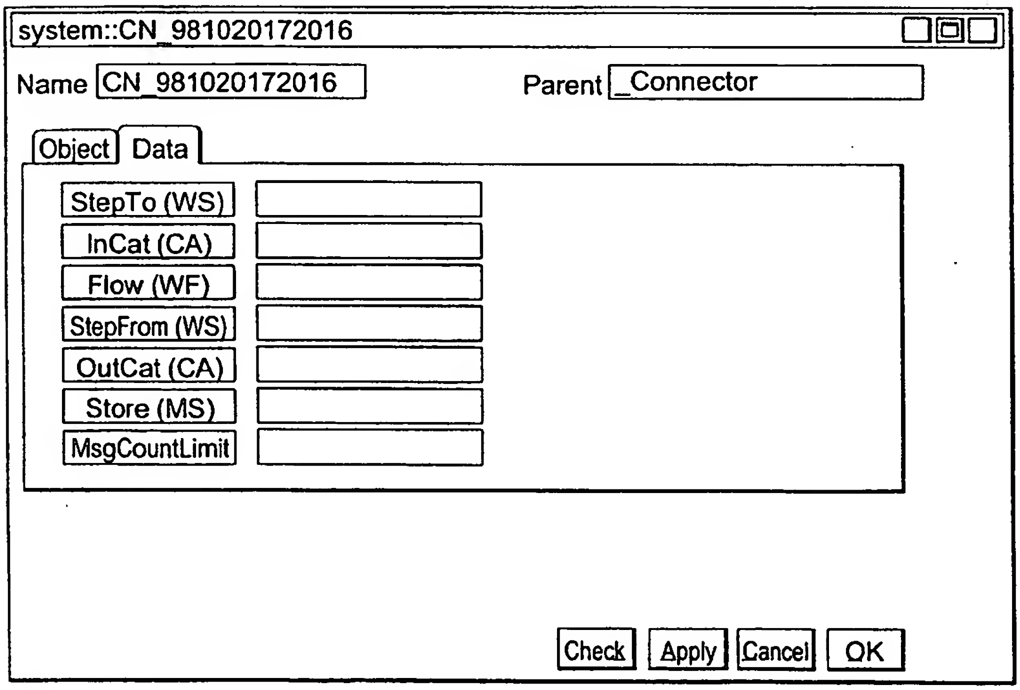


FIG. 20


system::CN_981020172016

Name CN_981020172016

Parent _Connector

Object | Data

StepTo (WS)

InCat (CA)

Flow (WF)

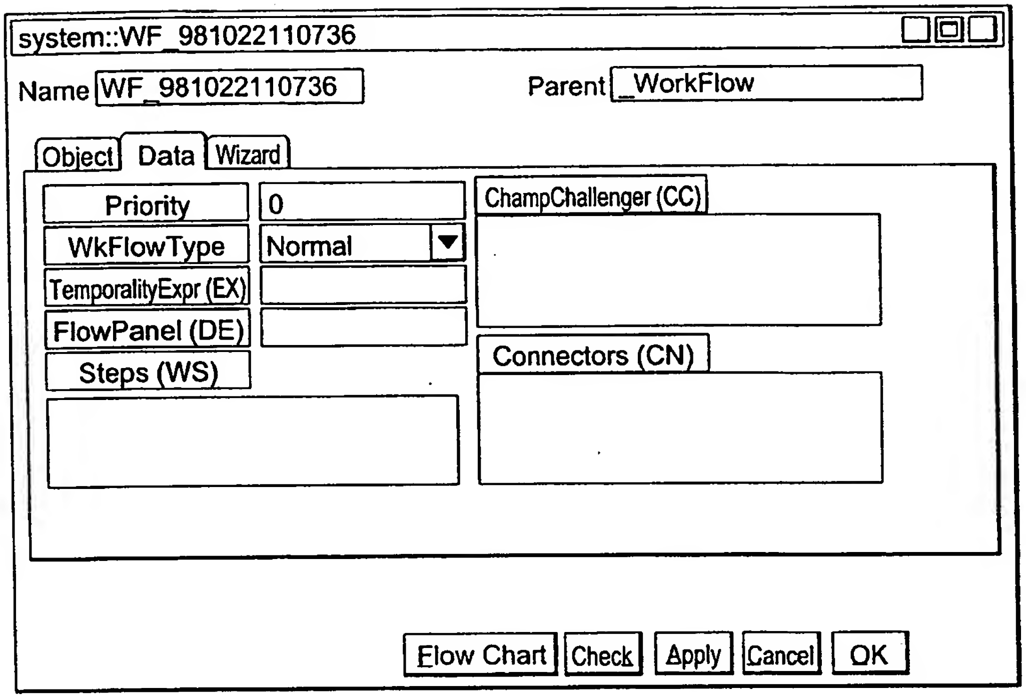
StepFrom (WS)

OutCat (CA)

Store (MS)

MsgCountLimit

FIG. 21



system::WF_981022110736

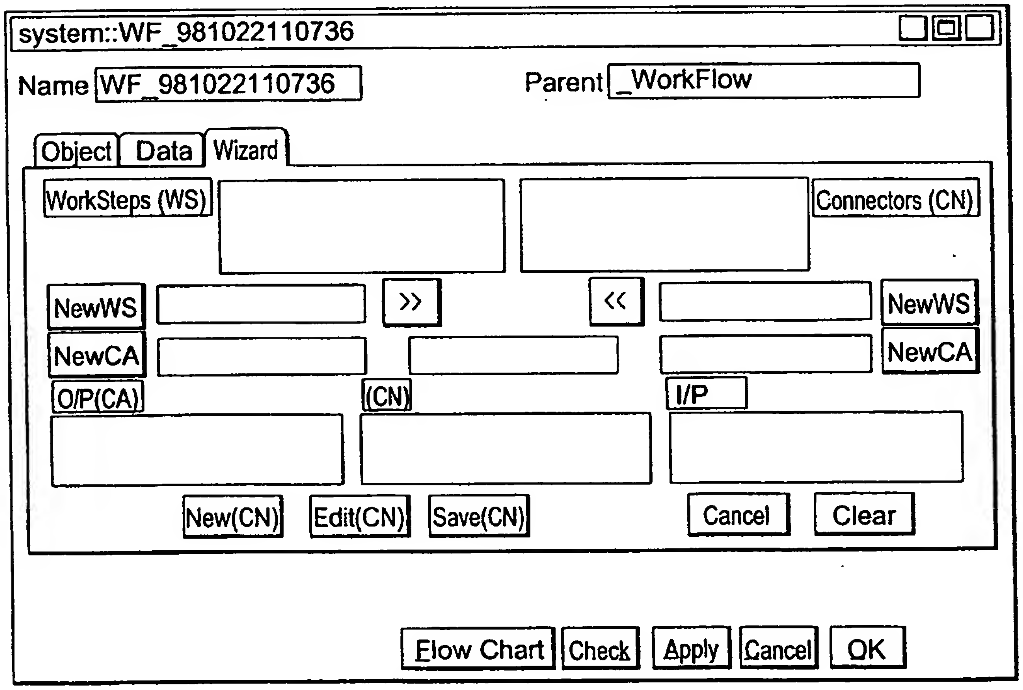
Name WF_981022110736

Parent _WorkFlow

Priority 0

WkFlowType Normal

FIG. 22A

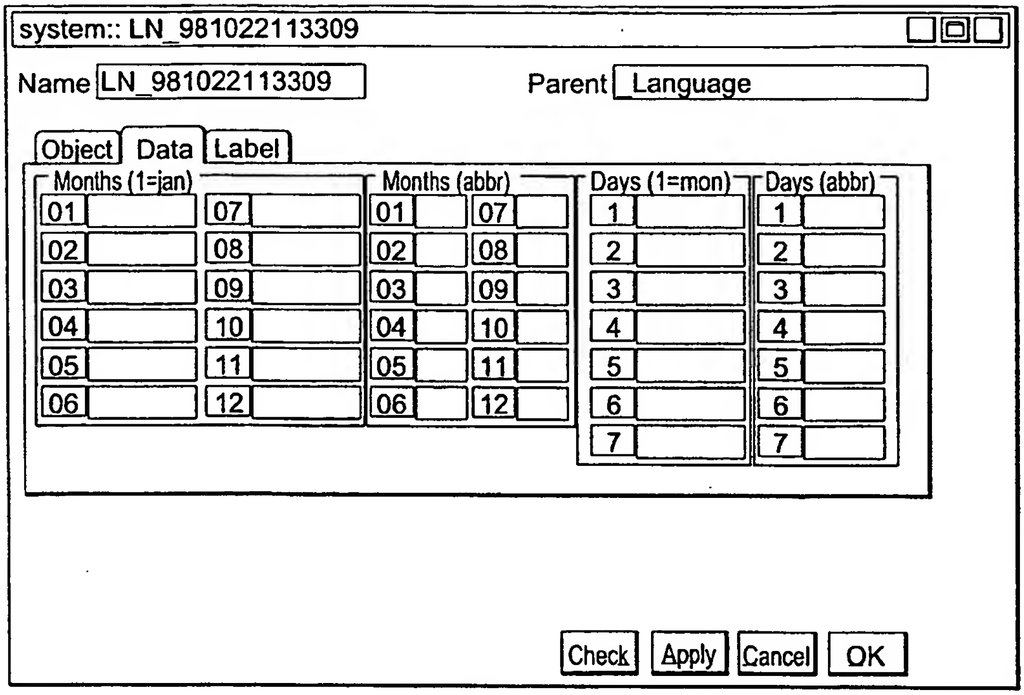


system::WF_981022110736

Name WF_981022110736

Parent _WorkFlow

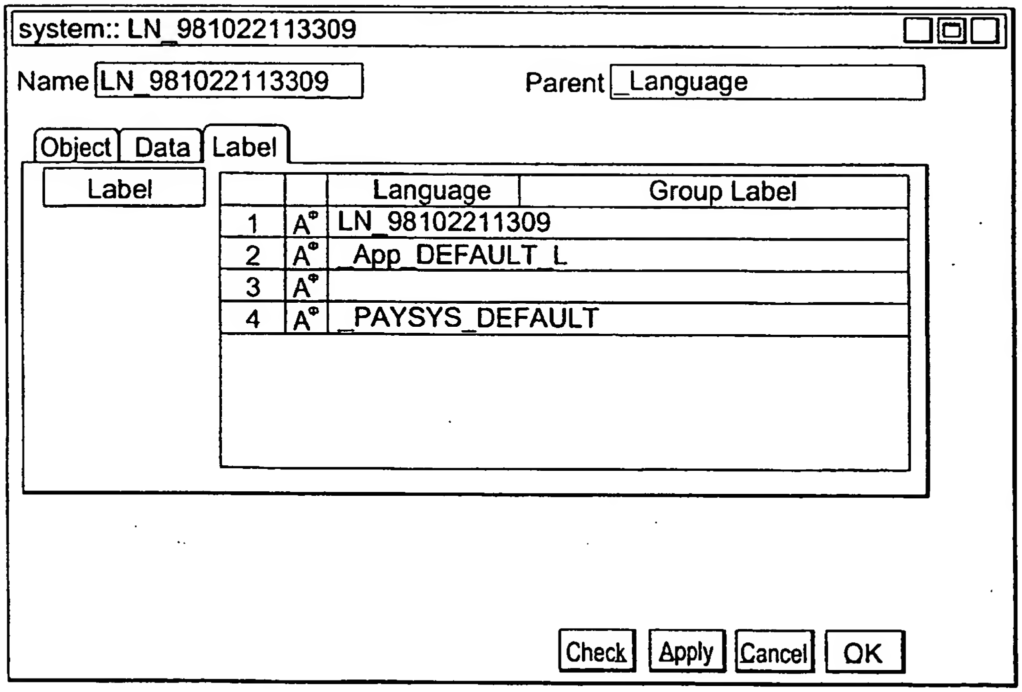
FIG. 22B



system:: LN_981022113309

Name LN_981022113309 Parent _Language

FIG. 23A

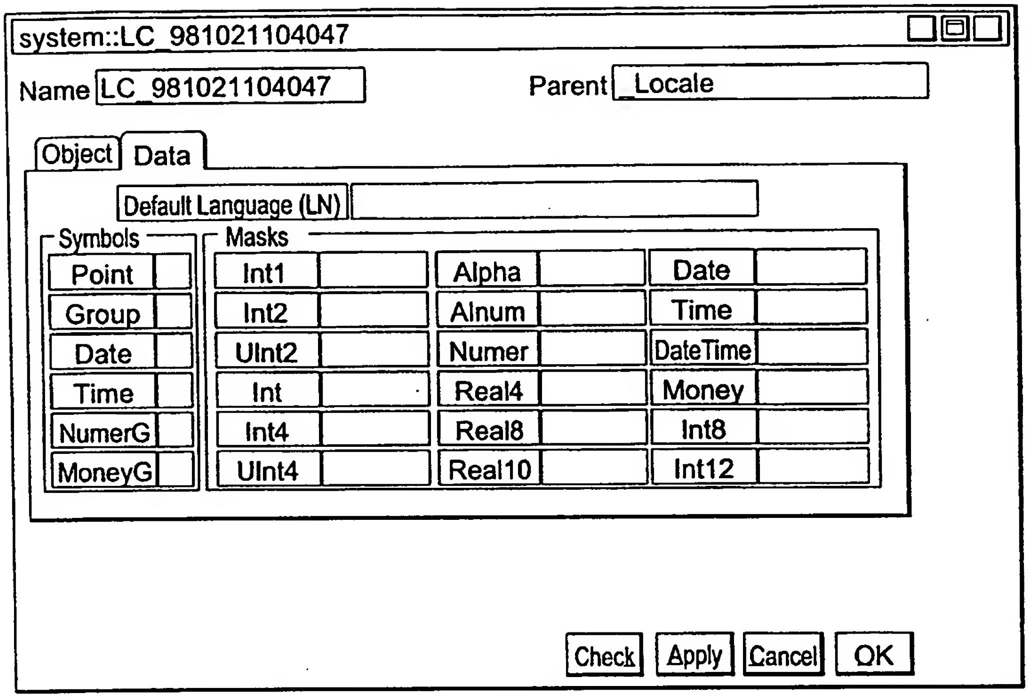


system:: LN_981022113309

Name LN_981022113309 Parent _Language

		Language	Group Label
1	A°	LN_98102211309	
2	A°	_App_DEFAULT_L	
3	A°		
4	A°	_PAYSYS_DEFAULT	

FIG. 23B



system::LC_981021104047

Name LC_981021104047

Parent _Locale

Object | Data

Default Language (LN)

Symbols

Point	
Group	
Date	
Time	
NumerG	
MoneyG	

Masks

Int1		Alpha		Date	
Int2		Alnum		Time	
UInt2		Numer		DateTime	
Int		Real4		Money	
Int4		Real8		Int8	
UInt4		Real10		Int12	

FIG. 24

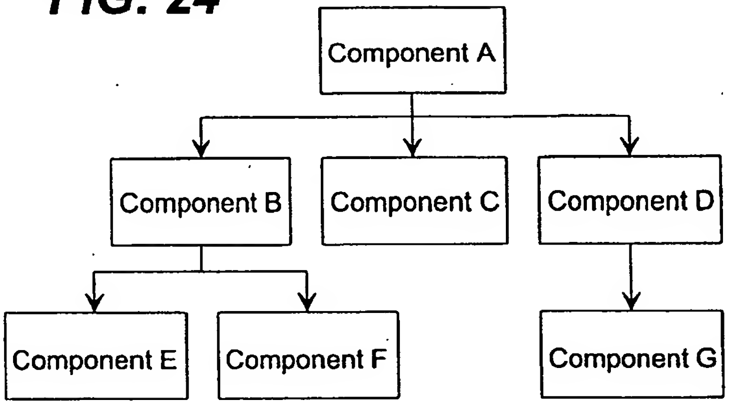


FIG. 25A

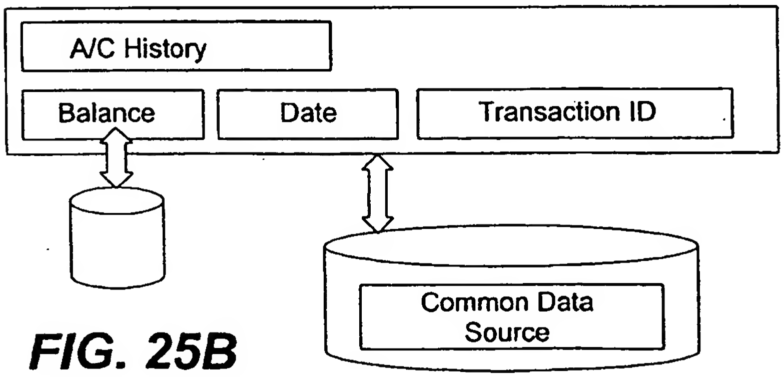
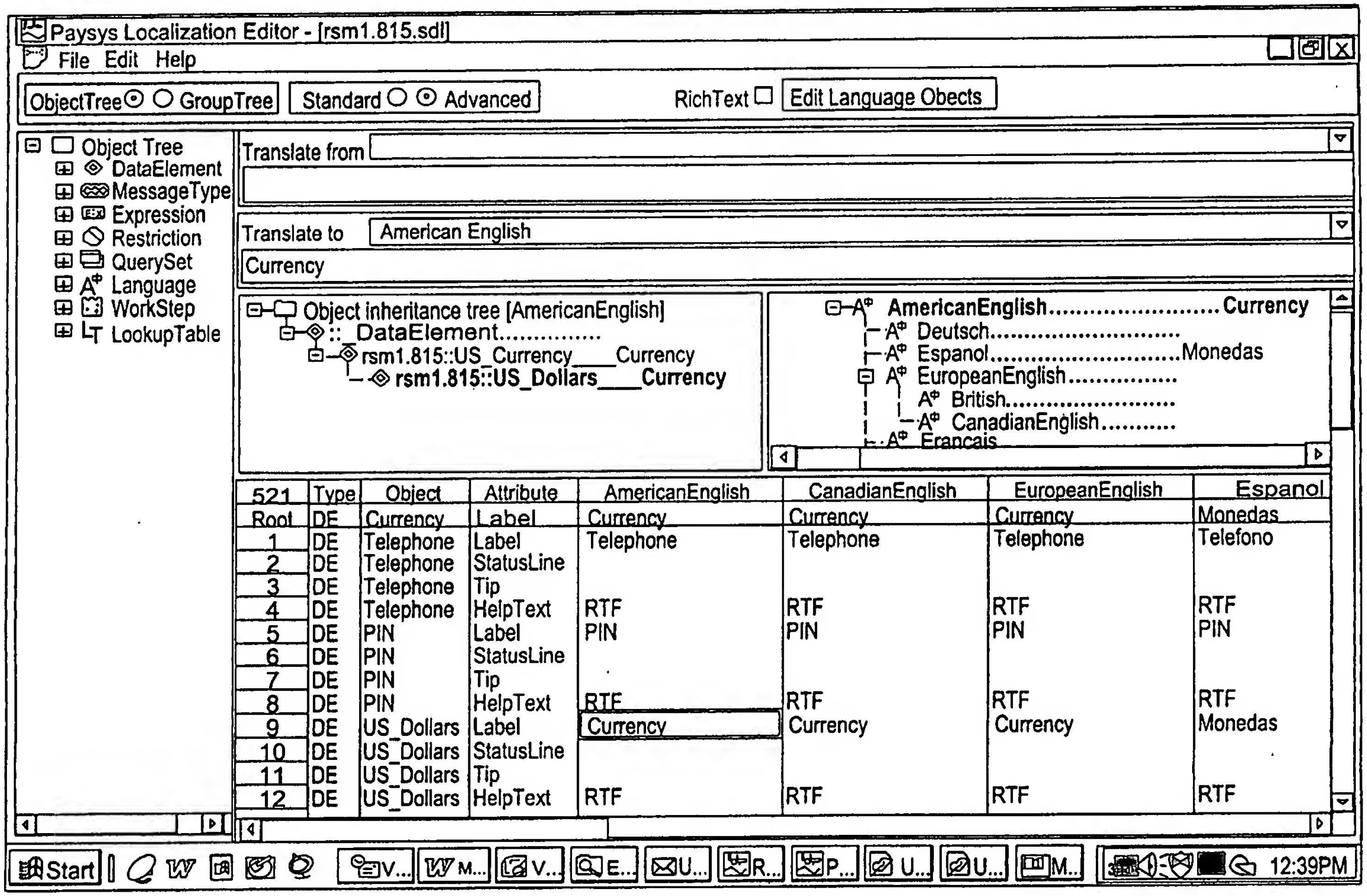


FIG. 25B


Paysys Localization Editor - [rsm1.815.sdl]
File Edit Help
Start
V... M... V... E... U... R... P... U... U... M...

ObjectTree ● ○ GroupTree

Standard ○ ● Advanced

RichText □

Edit Language Obects

Object Tree
- DataElement
- MessageType
- Expression
- Restriction
- QuerySet
- Language
- WorkStep
- LookupTable

Translate from

Translate to: American English

Currency

Object inheritance tree [AmericanEnglish]
- :: DataElement..............
  - rsm1.815::US_Currency____Currency
    - **rsm1.815::US_Dollars____Currency**

AmericanEnglish........................Currency
- Deutsch...........................
- Espanol...........................Monedas
- EuropeanEnglish...............
  - British........................
  - CanadianEnglish...........
- Francais

521	Type	Object	Attribute	AmericanEnglish	CanadianEnglish	EuropeanEnglish	Espanol
Root	DE	Currency	Label	Currency	Currency	Currency	Monedas
1	DE	Telephone	Label	Telephone	Telephone	Telephone	Telefono
2	DE	Telephone	StatusLine				
3	DE	Telephone	Tip				
4	DE	Telephone	HelpText	RTF	RTF	RTF	RTF
5	DE	PIN	Label	PIN	PIN	PIN	PIN
6	DE	PIN	StatusLine				
7	DE	PIN	Tip				
8	DE	PIN	HelpText	RTF	RTF	RTF	RTF
9	DE	US_Dollars	Label	Currency	Currency	Currency	Monedas
10	DE	US_Dollars	StatusLine				
11	DE	US_Dollars	Tip				
12	DE	US_Dollars	HelpText	RTF	RTF	RTF	RTF

12:39PM

FIG. 26

Account	"Inquiry"		
Account	"Deposit"	Amount	
Account	"Withdraw"	Amount	
Account	"Transfer"	Amount	Other Account

Data Elements

Name	DataType	FldType	FldLength
Account	Currency	ASCII	10
Type	Alpha	ASCII	10
Amount	Currency	ASCII	10
Other Account	Currency	ASCII	10

Message Types

Name	Elements	ParseParent	ParseSelector
Root	Account, Type	*None*	*None*
Inquiry	*None*	Root	Type == "Inquiry"
Common	Amount	Root	Type != "Inquiry"
Deposit	*None*	Common	Type == "Deposit"
Withdraw	*None*	Common	Type == "Withdraw"
Transfer	Other Account	Common	Type == "Transfer"

FIG. 27

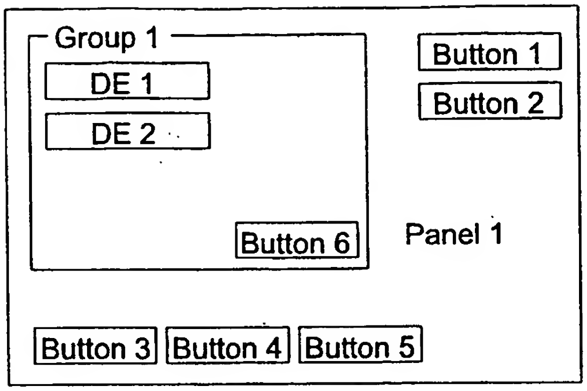


FIG. 28

	DE 1	DE 2	Button 6
Position	TopLeft	Relative	BottomRight
Hori(DE			
Vert(DE)		DE 1*	

FIG. 29

	Group1	Button 1	Button 2	Button 3	Button 4	Button 5
Position	TopLeft	TopRight	Relative	BottomLeft	Relative	Relative
Hori(DE					Button 3*	Button 4*
Vert(DE)			Button 1*			

FIG. 30

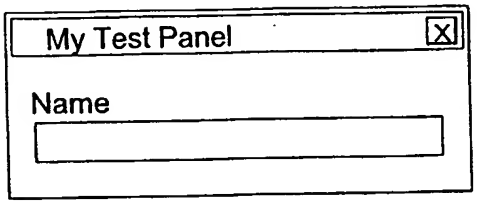


FIG. 31A

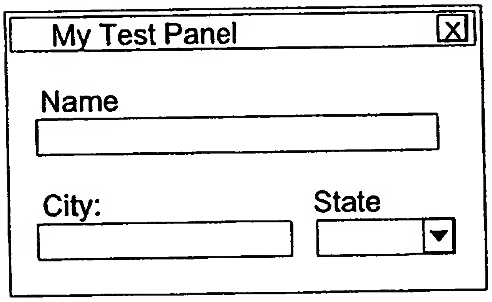


FIG. 31B

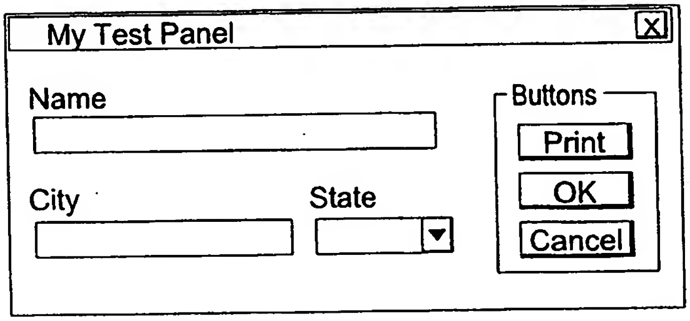
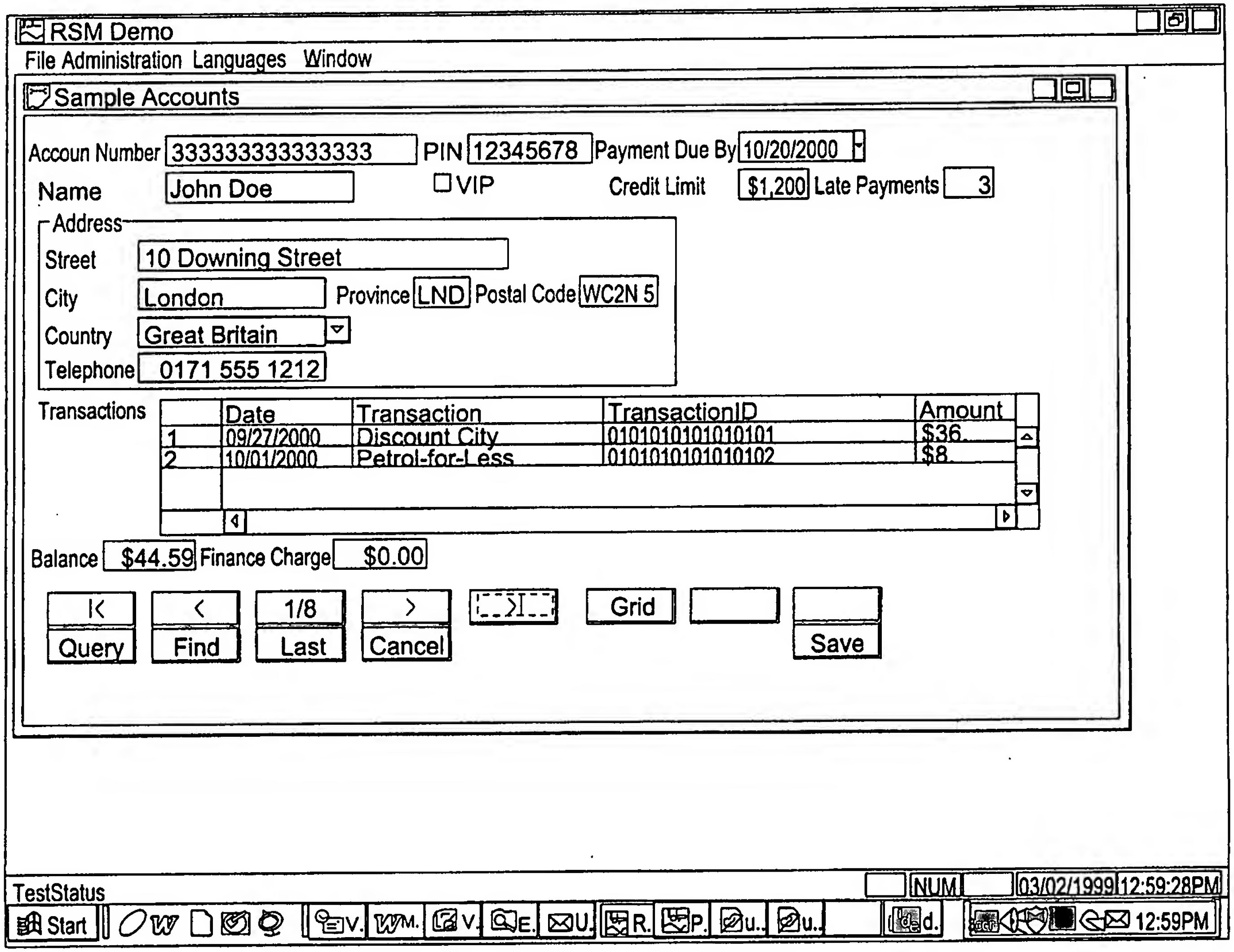


FIG. 31C



Accoun Number 333333333333333 PIN 12345678 Payment Due By 10/20/2000

Name John Doe □ VIP Credit Limit $1,200 Late Payments 3

Address

Street 10 Downing Street

City London Province LND Postal Code WC2N 5

Country Great Britain

Telephone 0171 555 1212

Transactions

	Date	Transaction	TransactionID	Amount
1	09/27/2000	Discount City	010101010101010101	$36.
2	10/01/2000	Petrol-for-Less	010101010101010102	$8.

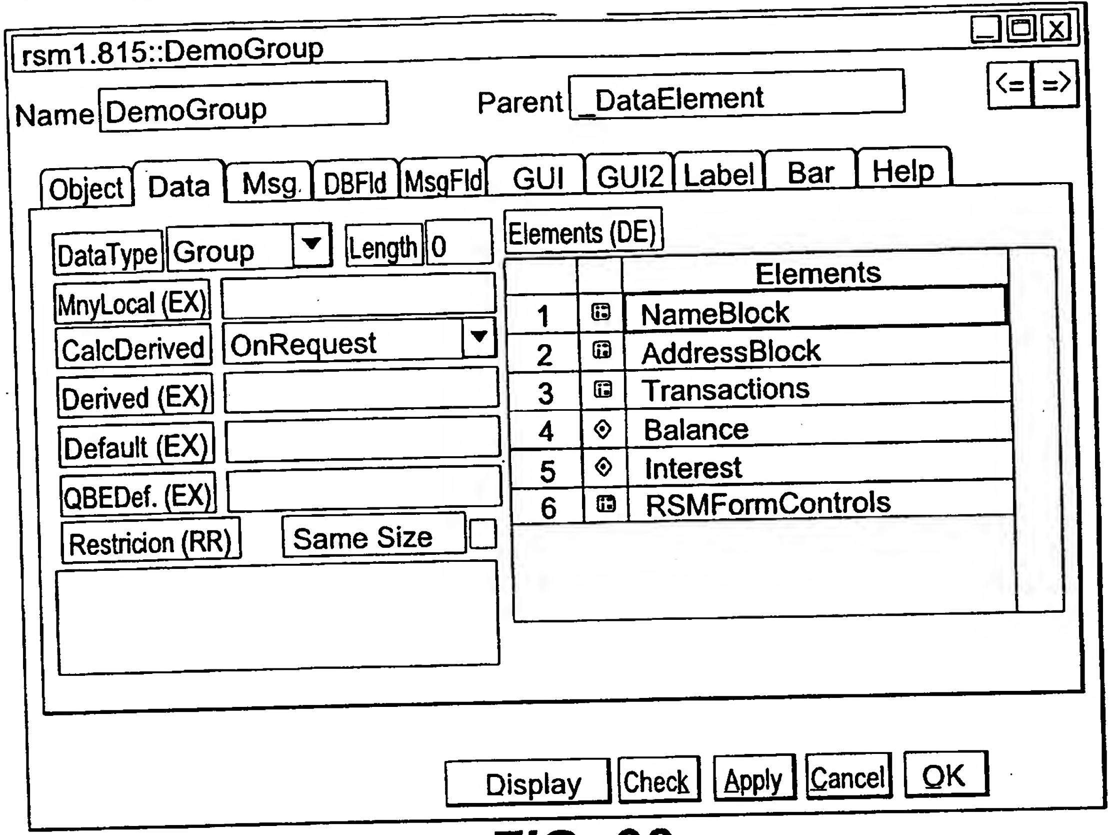
Balance $44.59 Finance Charge $0.00

1/8

03/02/1999 12:59:28PM

12:59PM

FIG. 32



rsm1.815::DemoGroup

Name: DemoGroup

Parent: _DataElement

DataType: Group

Length: 0

MnyLocal (EX):

CalcDerived: OnRequest

Derived (EX):

Default (EX):

QBEDef. (EX):

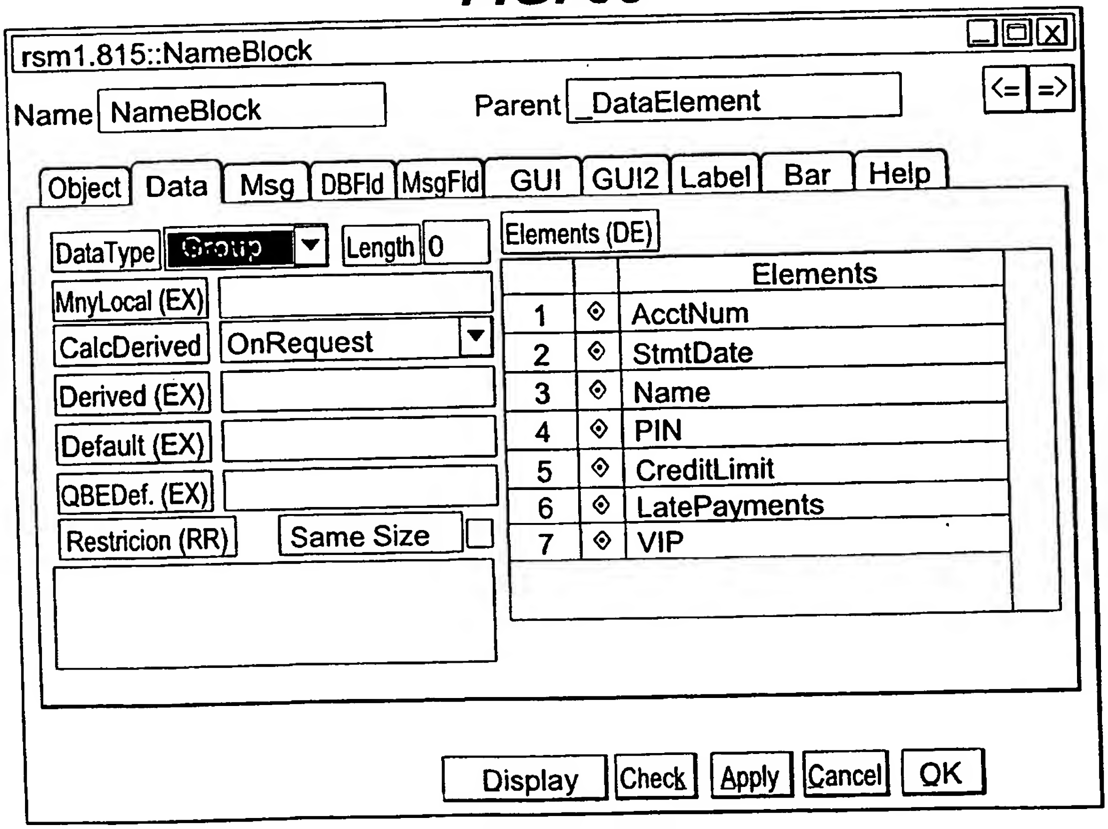
Restricion (RR)

Same Size □

Elements (DE)

		Elements
1	⊞	NameBlock
2	⊞	AddressBlock
3	⊞	Transactions
4	◈	Balance
5	◈	Interest
6	⊞	RSMFormControls

FIG. 33



rsm1.815::NameBlock

Name: NameBlock

Parent: _DataElement

DataType: Group

Length: 0

MnyLocal (EX):

CalcDerived: OnRequest

Derived (EX):

Default (EX):

QBEDef. (EX):

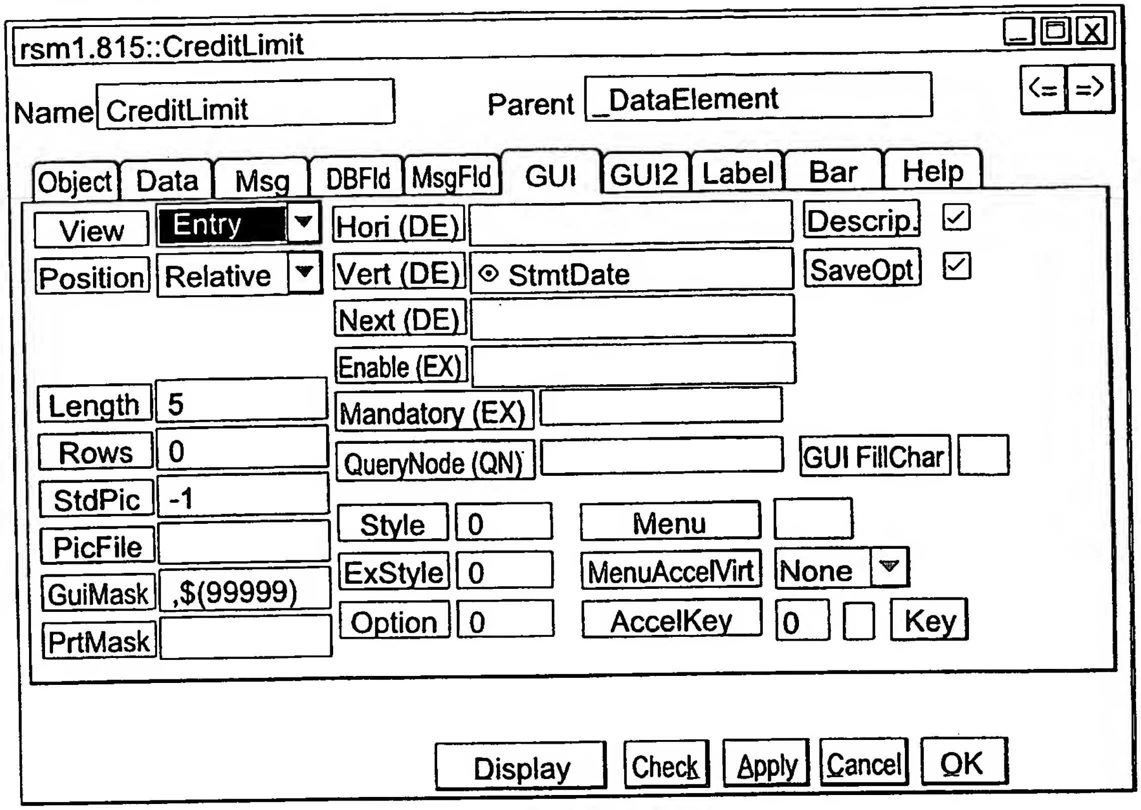
Restricion (RR)

Same Size □

Elements (DE)

		Elements
1	◈	AcctNum
2	◈	StmtDate
3	◈	Name
4	◈	PIN
5	◈	CreditLimit
6	◈	LatePayments
7	◈	VIP

FIG. 34



rsm1.815::CreditLimit

Name CreditLimit

Parent _DataElement

View Entry

Position Relative

Vert (DE) ⊙ StmtDate

Descrip. ☑

SaveOpt ☑

Length 5

Rows 0

StdPic -1

GuiMask ,$(99999)

Style 0

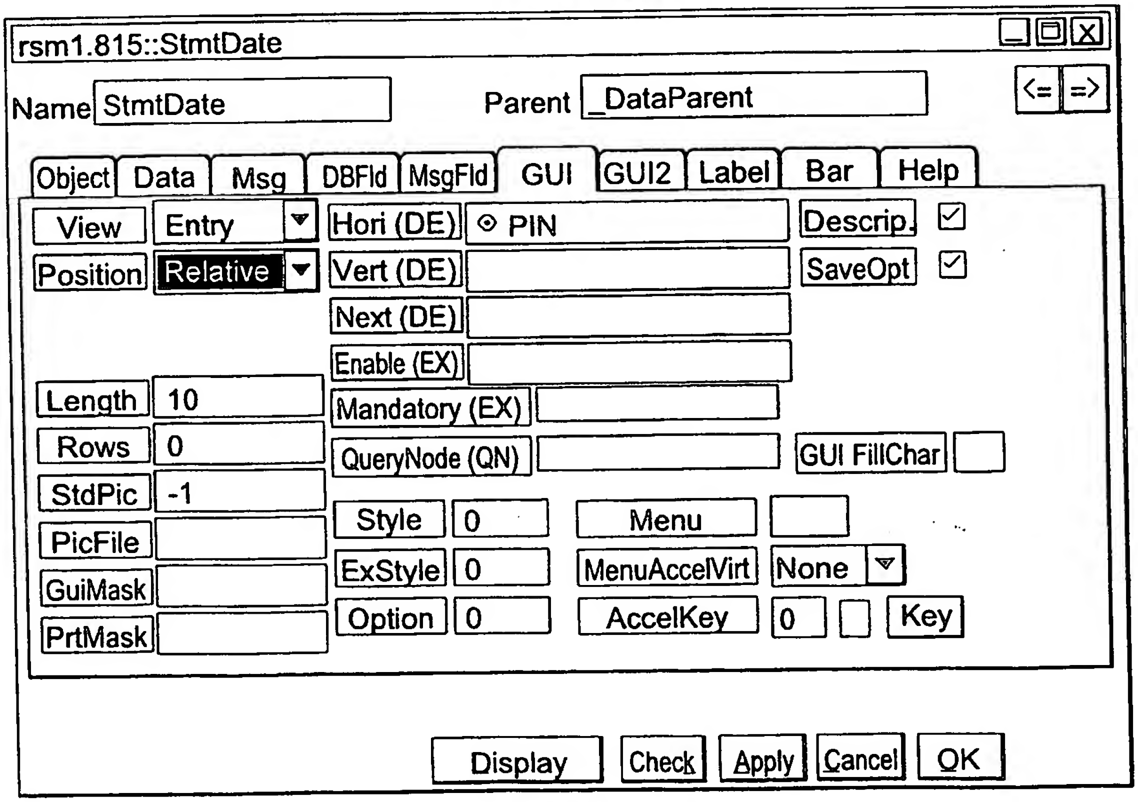
ExStyle 0

Option 0

MenuAccelVirt None

AccelKey 0 □

FIG. 35



rsm1.815::StmtDate

Name StmtDate

Parent _DataParent

View Entry

Position Relative

Hori (DE) ⊙ PIN

Descrip. ☑

SaveOpt ☑

Length 10

Rows 0

StdPic -1

Style 0

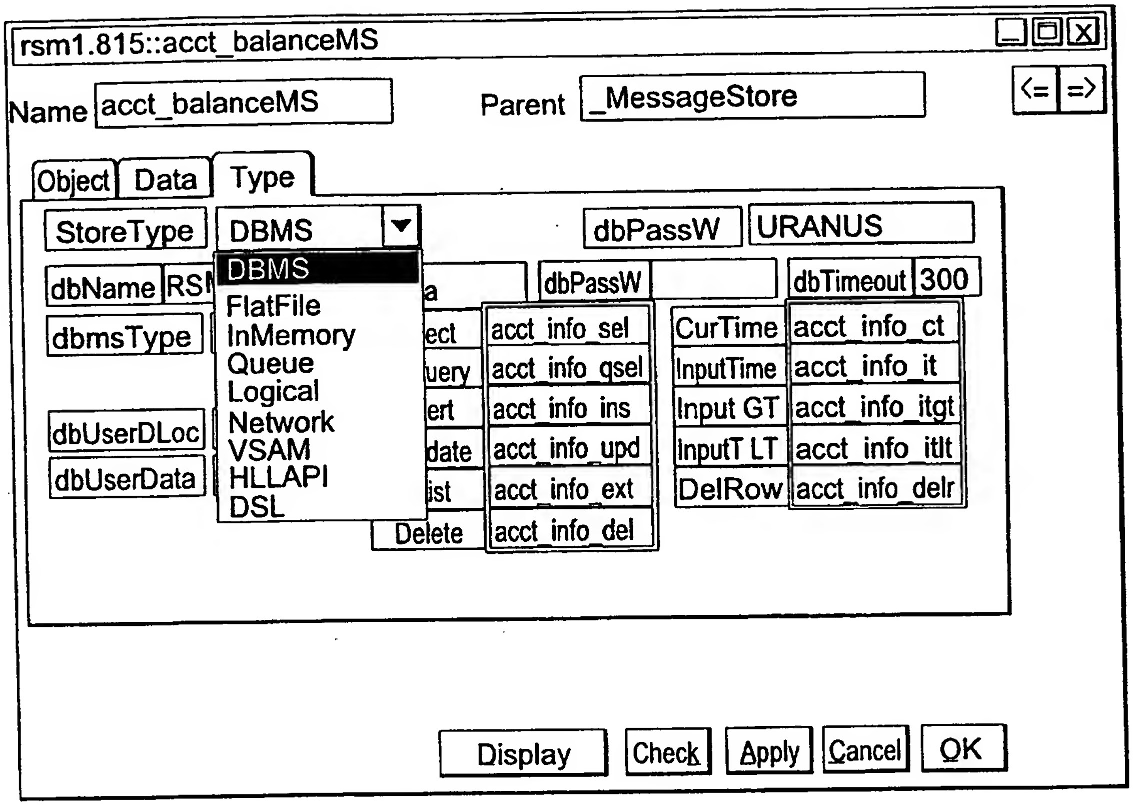
ExStyle 0

Option 0

MenuAccelVirt None

AccelKey 0 □

FIG. 36



rsm1.815::acct_balanceMS

Name acct_balanceMS

Parent _MessageStore

StoreType DBMS

dbPassW URANUS

dbName RSM

a

dbPassW

dbTimeout 300

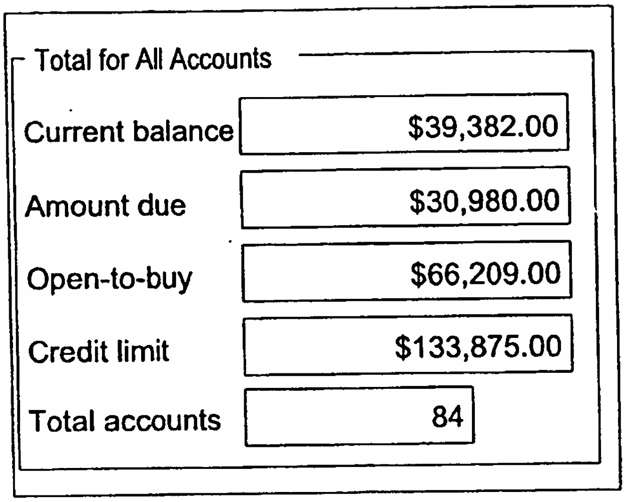
dbmsType

dbUserDLoc

dbUserData

ect	acct info sel	CurTime	acct info ct
uery	acct info qsel	InputTime	acct info it
ert	acct info ins	Input GT	acct info itgt
date	acct info upd	InputT LT	acct info itlt
ist	acct info ext	DelRow	acct info delr
Delete	acct info del		

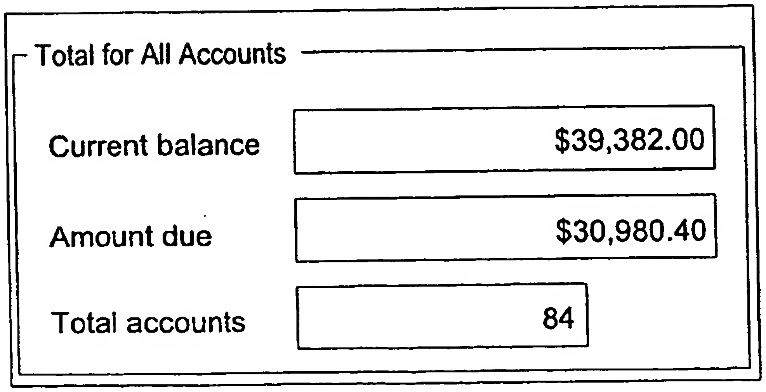
FIG. 37



Total for All Accounts

Current balance	$39,382.00
Amount due	$30,980.00
Open-to-buy	$66,209.00
Credit limit	$133,875.00
Total accounts	84

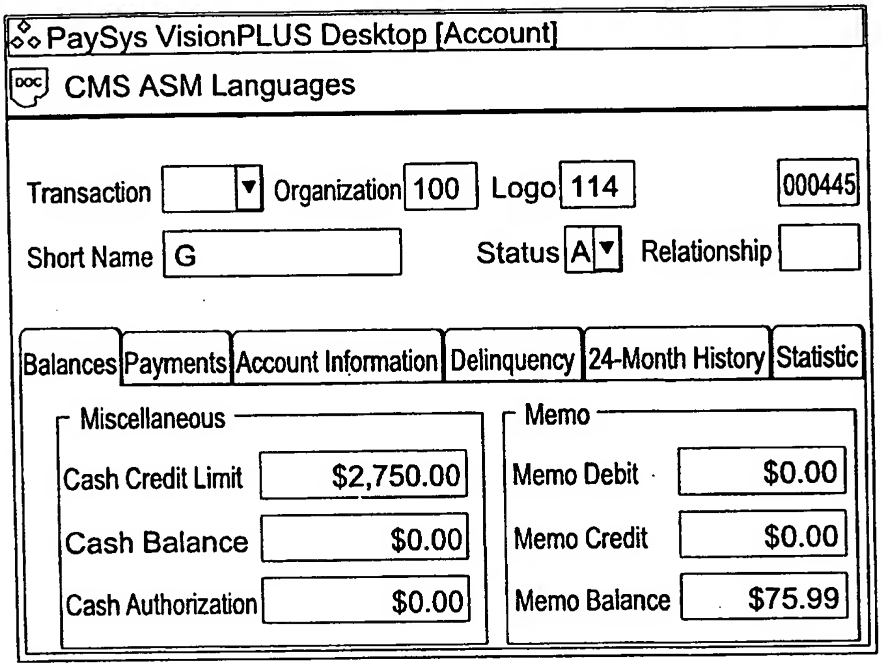
FIG. 38A



Total for All Accounts

Current balance	$39,382.00
Amount due	$30,980.40
Total accounts	84

FIG. 38B



PaySys VisionPLUS Desktop [Account]

CMS ASM Languages

Transaction ▼ Organization 100 Logo 114 000445

Short Name G Status A ▼ Relationship

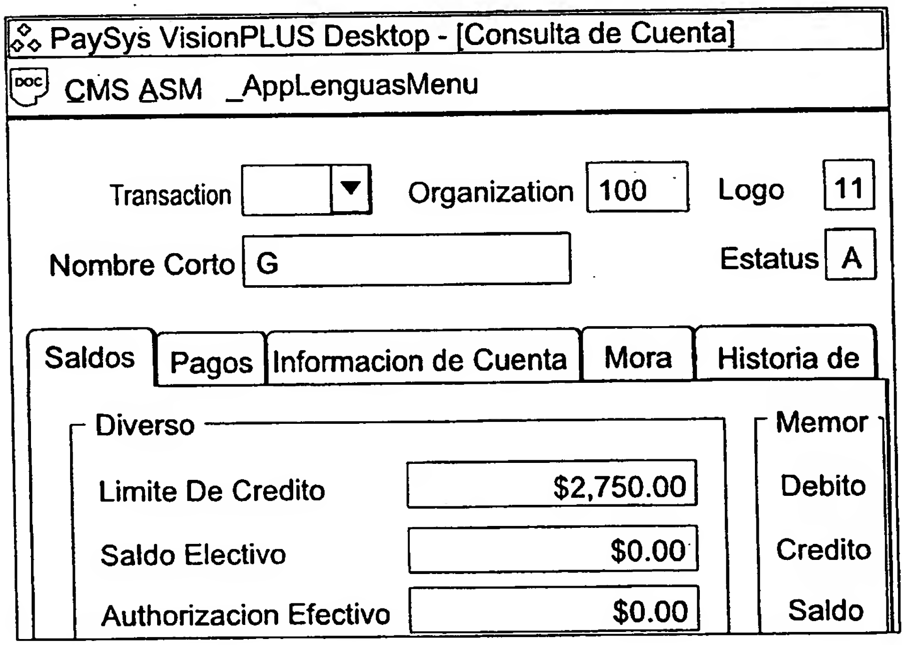
Miscellaneous

Cash Credit Limit	$2,750.00
Cash Balance	$0.00
Cash Authorization	$0.00

Memo

Memo Debit	$0.00
Memo Credit	$0.00
Memo Balance	$75.99

FIG. 39A



PaySys VisionPLUS Desktop - [Consulta de Cuenta]

CMS ASM _AppLenguasMenu

Transaction ▼ Organization 100 Logo 11

Nombre Corto G Estatus A

Diverso

Limite De Credito	$2,750.00
Saldo Electivo	$0.00
Authorizacion Efectivo	$0.00

Memor

Debito

Credito

Saldo

FIG. 39B

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
14 September 2000 (14.09.2000)

PCT

(10) International Publication Number
WO 00/54202 A3

(51) International Patent Classification⁷: G06F 17/60

(21) International Application Number: PCT/US00/06388

(22) International Filing Date: 10 March 2000 (10.03.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/123,976 11 March 1999 (11.03.1999) US

(71) Applicant *(for all designated States except US)*: **PAYSYS INTERNATIONAL, INC.** [US/US]; One Meca Way, Norcross, GA 30093 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: **BLACK, David, B.** [US/US]; 41 Clinton Avenue, Ridgewood, NJ 07450 (US). **MORGAN, Michael, J.** [US/US]; 4220 River Bottom Drive, Norcross, GA 30092 (US). **BHAGAT, Rohit** [IN/US]; Apartment 1621, 3450 Breckenridge, Duluth, GA 30096 (US).

(74) Agents: **PRATT, John, S.** et al.; Kilpatrick Stockton LLP, Suite 2800, 1100 Peachtree Street, Atlanta, GA 30309-4530 (US).

(81) Designated States *(national)*: AE, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

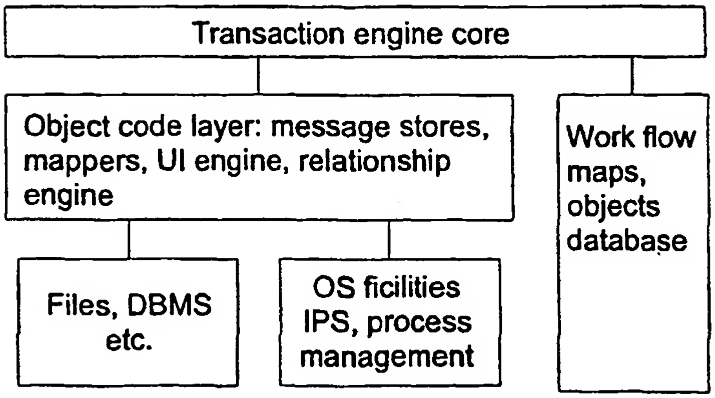
(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*
— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments*

(88) Date of publication of the international search report: 13 June 2002

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: METHODS AND SYSTEMS FOR DEVELOPING APPLICATIONS AND FOR INTERFACING WITH USERS


Transaction engine core
Object code layer: message stores, mappers, UI engine, relationship engine
Work flow maps, objects database
Files, DBMS etc.
OS ficilities IPS, process management

(57) Abstract: A declarative approach to programming involves providing a transaction engine and a repository. The transaction engine controls the behavior of an application by routing messages between a pre-defined group of work steps. The repository includes a declaration space within which components are defined as well as the relationship between the components. An administrator includes an Integrated Development Environment (IDE) editor for allowing developers to access and modify portions of the repository. In the preferred embodiment, developers are allowed to define attributes of data elements, define relationships between the data elements, and to organize data elements into messages through the repository. Furthermore, developers are able to define the logical-to-physical mapping in the repository, such as by defining message stores. An application is defined by a workflow comprised of a number of worksteps interconnected to each other. The workflow and the worksteps are also defined within the repository. Interfaces developed with the repository and transaction engine can be dynamically changed based on user, security, language, and locale. The repository and transaction engine also enable the inheritance of values and provides message inheritance, language inheritance, sibling inheritance, container inheritance, and attribute inheritance.

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 00/06388

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F17/60

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

EPO-Internal, IBM-TDB, INSPEC

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	SARIN S K: "Object-oriented workflow technology in InConcert" DIGEST OF PAPERS OF COMPCON (COMPUTER SOCIETY CONFERENCE) 1996 TECHNOLOGIES FOR THE INFORMATION SUPERHIGHWAY. SANTA CLARA, FEB. 25 - 28, 1996, DIGEST OF PAPERS OF THE COMPUTER SOCIETY COMPUTER CONFERENCE COMPCON, LOS ALAMITOS, IEEE COMP. SOC. PRESS, , vol. CONF. 41, 25 February 1996 (1996-02-25), pages 446-450, XP010160935 ISBN: 0-8186-7414-8 page 446, right-hand column, paragraph 2 -page 449, left-hand column, paragraph 1 --- -/--	1-42

☒ Further documents are listed in the continuation of box C.

☒ Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search 3 April 2002	Date of mailing of the international search report 09/04/2002
Name and mailing address of the ISA European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Authorized officer Daman, M

2

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 00/06388

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
Y	US 5 604 896 A (DUXBURY PAUL ET AL) 18 February 1997 (1997-02-18) abstract column 1, line 61 -column 2, line 56 figure 1 ---	1-42
A	HSU M ET AL: "WORK-FLOW AND LEGACY SYSTEMS ADDING WORK-FLOW SUPPORT WILL BECOME CRITICAL TO LEGACY TRANSACTION-PROCESSING APPLICATIONS" BYTE, MCGRAW-HILL INC. ST PETERBOROUGH, US, vol. 19, no. 7, 1 July 1994 (1994-07-01), pages 109-110,112,114, XP000445519 ISSN: 0360-5280 page 109, right-hand column, paragraph 1 -page 114, right-hand column, paragraph 2 ---	1,21,30
A	WO 94 18620 A (ACTION TECH INC) 18 August 1994 (1994-08-18) abstract page 1, paragraph 1 -page 5, paragraph 1 -----	1,21,30

2

INTF¬NATIONAL SEARCH REPORT

Information on patent family members

Interr. nal Application No
PCT/US 00/06388

Patent document cited in search report		Publication date	Patent family member(s)		Publication date
US 5604896	A	18-02-1997	DE	69421125 D1	18-11-1999
			DE	69421125 T2	31-05-2000
			EP	0634718 A2	18-01-1995
WO 9418620	A	18-08-1994	AU	6133594 A	29-08-1994
			EP	0686282 A1	13-12-1995
			JP	9501517 T	10-02-1997
			WO	9418620 A1	18-08-1994
			US	6073109 A	06-06-2000